Guidelines for combination of static and dynamic analyses

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 / D3.3 / 1.0
Work package contributing to the deliverable:	WP 3
Due date:	December 2018 – M24
Actual submission date:	13th December 2018

Responsible organisation:	DA
Editor:	D. PARIENTE
Dissemination level:	PU
Revision:	1.0

Abstract:	This report is the output of task 3.2 and contains guidelines for combination of static and dynamic analyses.
Keywords:	static analysis, dynamic analysis, test, concolic technique, fuzzing, vulnerability

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)

Contributors (ordered according to beneficiary numbers)

Mounir KELLIL (CEA)

Julien SIGNOLES (CEA)

Jens GERLACH (FOKUS)

Allan BLANCHARD (INRIA)

Executive Summary

This report presents the results of task 3.2 in WP3.

In complement to static and dynamic tool coupling activities covered by task 2.4, and in order to expose the current state-of-the-art of the technology related to VESSEDIA expected outcomes, we detail in this task the methodological aspects of combined static and dynamic analyses. In particular, it is organized as a guideline for the coupling of verification analyses and testing techniques in order, for instance, either to find vulnerabilities, or even demonstrate the absence of vulnerabilities, or generate useful counter-example scenarios violating sought security properties.

This includes the identification of complementary relations between some static and dynamic techniques, and their most appropriate integration approaches. For example, testing by fuzzing techniques can be used to remove false positives or illustrate potential security holes, based on an abstract-interpretation based analysis and intensive testing.

A supplement in Annex 1 presents an innovative approach developed in the project, introducing a new notion of verification coverage that permits to combine test and proof for verifying a group of specifications related to the same piece of code, subsuming both the usual notions of functional coverage and structural coverage.

Another supplement in Annex 2 gives an illustration and proposes a reflection about the positioning of testing and formal analysis, in particular with regards to the Frama-C toolbox. In most cases, both approaches - testing and verification - need efforts to validate requirements, if taken separately, but combining them gives some opportunity to promising gains in terms of efficiency.

Contents

List of Figures

Chapter 1 Introduction

Static analysis of source code encompasses several approaches, all of them sharing the same essential feature: the code under analysis is never executed (no runtime), but interpreted by formal means or patterns or any heuristics extracting valuable information from the source code.

At the opposite, dynamic analysis on a given application relies on executions, sometimes deeply instrumented or just outfitted by observers, which provides complementary information.

Roughly presented, in practice, static analysis covers more cases: application inputs are not taken into account one-by-one but as a mathematical set of values processed at once (completeness). This is often performed at the price of over-approximations (imprecision), and weak ability to scale up, even if impressive progress was done in the recent years. On the contrary, dynamic analysis is, by definition, able to deal with an entire application generally executed in its targeted environment with precise results. But the drawback is that, in most cases, only a very few application input domain can be taken into account (incompleteness): typically, testing is basically based on a limited input domain sampling and therefore does not permit to cover the entire input domain, also known as combinatorial explosion.

Namely, coupling static and dynamic analyses aims at mutually palliating their limitations when considered separately.

In the following, we present some of these couplings, essentially based on VESSEDIA tools and methods. They are not exhaustive as, by experience, every industrial verification objective and context may harbour new potential usage of combined static and dynamic techniques, or may not be compliant with some methodological constraints. This report is therefore intended to be an incentive to these forthcoming couplings, as most of the underlying tools and techniques may be still in progress. It is also worth mentioning that no complete methodology for coupling all of the VESSEDIA plugins together is currently available; however other deliverables (D1.5 (M24), and D2.4 (M32) should try to unify them in some promising ways.

Last, the document in Annex 2 (an experimentation on testing and verification) provides an illustration and a reflection about the positioning of testing and formal analysis, in particular with the Frama-C toolbox. These considerations are illustrated with a C++ library function code.

Other state-of-the-art

For a complementary and brief state-of-the-art, but unrelated to VESSEDIA activities and outcomes, we recommend the reading of [5] (concept of residual investigation, dynamic invariant inference, and generating example data of dataflow programs using dynamic symbolic execution in order to achieve high coverage), and [8] based on a taint analysis and using genetic algorithm to generate inputs for the program under analysis. These approaches could be to some extent an interesting complement to these guidelines which were only focused on VESSEDIA approaches, and which could be in no way exhaustive due to the large perimeter of potentially involved static & dynamic paradigms, methods and tools.

Chapter 2 Coupling methods and approaches

This section reminds the Frama-C platform and its main components (2.1), and the different coupling methods and other approaches combining static and dynamic tools, essentially based on VESSEDIA tools.

For each combination, we expose the major objectives and the main limitations.

2.1 Frama-C, a platform for analysis of C code

Frama-C is a platform dedicated to the analysis of source code written in C. The Frama-C platform gathers several analysis techniques into a single collaborative extensible framework. The collaborative approach of Frama-C allows analyzers to build upon the results already computed by other analyzers in the framework. Thanks to this approach, Frama-C can provide a number of sophisticated tools such as an abstract interpreter, a weakest precondition calculus procedure, a slicer, and other static analyzers allowing further source code analyses.

Frama-C has some major differences with classical heuristic-based analysis tools: it aims at being correct, that is, never to remain silent for a location in the source code where an error can happen at run-time. It also allows its user to manipulate functional specifications, and to prove that the source code satisfies these formal specifications.

Heuristic bug-finding tools can be very useful, but because they do not find all bugs, they cannot be used to prove (in the mathematical sense) the total absence of bugs in a program. Frama-C on the other hand can guarantee that there are no bugs in a program (meaning no run-time error, and no deviation from the functional specification the program is supposed to meet). This of course requires more work from the user than heuristic bug-finding tools usually do, but some of the analyses provided by Frama-C require comparatively little intervention from the user, and the collaborative approach proposed in Frama-C allows the user to get results about complex semantic properties.

In the following chapters, the presented tools rely on the Frama-C platform, illustrating its capability to be extended in two directions: more subtle and in-depth static analyses, and a powerful and fruitful extensible and collaborative tool allowing cooperation between static and dynamic approaches.

The analysis plug-ins distributed together with Frama-C are numerous. They are briefly presented below as an appetizer of the platform capabilities, or a brief preliminary reminder introducing the coupling methods exposed later in this report.

Static analysis plug-ins

Evolved Value analysis (EVA)

This plug-in[1] computes variation domains for the variables in the source code. It is quite automatic, although the user may guide the analysis in places. It handles a wide spectrum of C constructs. This plug-in uses abstract interpretation techniques: input domains are propagated through the code, and runtime error alarms – if any – are raised accordingly.

1 https://frama-c.com/download/frama-c-value-analysis.pdf

WP

This plug-in is based on weakest precondition computation techniques[2]. It allows proving that C functions satisfy their specification as expressed in ACSL. These proofs are modular: the specifications of the called functions are used to establish the proof without looking at their code. This modularity permits in some way a certain scalability.

Mthread

The Mthread plug-in[3] automatically analyzes concurrent C programs, using the techniques used by the EVA plug-in. At the end of its execution, the concurrent behaviour of each thread is over-approximated. Thus, the information delivered by the plug-in takes into account all the possible concurrent behaviours of the program under analysis.

Dynamic-oriented analysis plug-ins

E-ACSL

E-ACSL plug-in[4] translates annotations into C code for runtime assertion checking (E-ACSL is presented later more in details in this document).

PathCrawler

PathCrawler's principal functionality[5] is to automate structural unit testing by generating test inputs which cover all the feasible execution paths for the C function under test. It can also be used to satisfy other coverage criteria (like *k-path* coverage restricting the all-path criterion to paths with at most *k* consecutive loop iterations, branch coverage, MC-DC,...), to generate supplementary tests to improve coverage of an existing functional test suite or to generate just the tests necessary to cover the part of the code which has been impacted by a modification.

Annotation generation plug-ins

Aoraï

Aoraï is a Frama-C plug-in[6] which provides a method to automatically annotate a C program according to an automaton *F* such that, if the annotations are verified, one ensures that the program respects *F*. A classical method to validate annotations then is to use the Jessie plugin and the Why tool or the WP plug-in.

RTE

The runtime error annotation plug-in RTE (incorporated in WP) mainly allows the generation of annotations for common runtime errors (in the sense of undefined behaviours of ISO C99), and overflows on unsigned integer operations (which is indeed a well-defined behaviour).

[2] https://frama-c.com/download/frama-c-wp-manual.pdf

[3] https://frama-c.com/download/frama-c-mthread-manual.pdf

[4] https://frama-c.com/download/e-acsl/e-acsl-manual.pdf

[5] https://frama-c.com/download/frama-c-pathcrawler.pdf

[6] https://frama-c.com/download/frama-c-aorai-manual.pdf

Code comprehension and other manipulations

Impact analysis

This plug-in highlights the locations in the source code that are impacted by a modification. This plug-in depends on the results obtained by the EVA plug-in.

Scope & Data-flow browsing

This plug-in allows the user to navigate the dataflow of the program, from definition to use or from use to definition. Scope plug-in only takes into account the executions that have been considered by the EVA plug-in.

Variable occurrence browsing

Also provided as a simple example for new plug-in development (final users can indeed develop their own analyzers based on existing Frama-C analyses), this plug-in allows the user to reach the statements where a given variable is used. This plug-in also depends on results of the EVA plug-in.

Metrics calculation

This plug-in[7] allows the user to compute various metrics from the source code. It is based on the results computed by the EVA plug-in.

Semantic constant folding

This plug-in makes use of the results of the EVA plug-in to replace, in the source code, the constant expressions by their values. Because it relies on EVA, it is able to do more of these simplifications than a syntactic analysis would.

Slicing

This plug-in (based on EVA) slices the code according to a user-provided criterion: it creates a copy of the program which can be compiled, but keeps (an over-approximation of) those parts which are necessary with respect to the given criterion (documented with the EVA plug-in).

Spare code

Remove "spare code", i.e. code that does not contribute to the final results of the program. It is also relying on EVA computations.

2.2 Detecting runtime errors by static analysis and testing (SANTE)

SANTE (Static ANalysis and TEsting) is combining value analysis (EVA plug-in), program slicing and structural testing for the verification of C programs [9].

The method uses EVA to report alarms of possible run-time errors (some of which may be false alarms), and proposes test generation to confirm or to reject them. The method produces for each alarm a diagnostic that can be safe for a false alarm, bug for an effective bug confirmed by some input state, or unknown if it does not know whether this alarm is an effective error or not. Experimental results shows that the combined method is better than each technique used independently. It is more precise than a static analyzer and more efficient in terms of time and number of detected bugs than a concolic structural testing tool used alone, or even guided by the exhaustive list of alarms for all potentially threatening statements.

A program slicing step is also included into the SANTE process, in order to simplify and reduce the source code before test generation. Program slicing is a technique for decomposing programs based

[7] https://frama-c.com/download/frama-c-metrics-manual.pdf

on data and control-flow information with respect to a given slicing criterion (e.g. one or several program statements). The slicing process can be applied either to the whole set of alarms at a glance (obtaining one unique slice), or to each alarm thus generating one source code slice per alarm, depending on the nature of the original program and its alarms.

Finally, dynamic analysis will permit to debug the simplified program. Indeed, it is clear cut that dynamic analysis will run much faster (depending on the program) than on the original code, as it is applied to a simplified (sliced) version of the code.

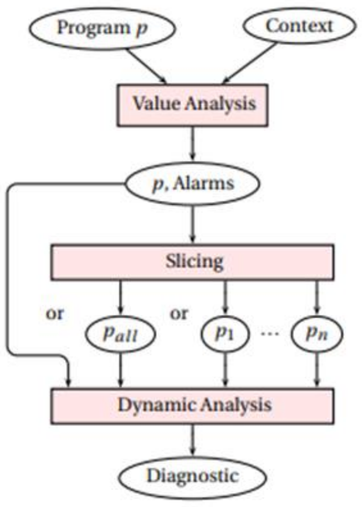


Figure 1: SANTE tool process

SANTE only handles a small subset of ACSL specifications (assertions preventing divisions by zero, out-of-bounds array accesses and arithmetic overflows), and does not share the results of the test generation with other plugins. The need to extend the support of ACSL constructions is one of the main motivations that led to a new plugin, STADY (presented in the next section), which generalizes the SANTE method and overcomes most of its limitations.

2.3 Deductive verification assisted by testing (STADY)

The objective of this tool [12] is to facilitate the design of combined static analysis (S/A) and dynamic analysis (D/A) in Frama-C. On the one hand, such combinations allow any S/A plug-in to reuse the results of test generation, allowing the D/A to help the S/A and conversely. On the other hand, during deductive verification of C programs, test generation can automatically provide the validation engineer with a fast and helpful feedback facilitating the verification task.

For example, while specifying a C program, test generation may find a counter-example showing that the current specification does not hold for the current code. In case of a proof failure for a specified program property during program proof, when the validation engineer has no other alternative than manually analyzing the reasons of the failure, test generation can be particularly useful.

The absence of counter-examples after a rigorous partial (or, when possible, complete) exploration of program paths provides additional confidence in (respectively, guarantee of) the correctness of the program with respect to its current specification. This feedback may encourage the engineer to think that the failure is due to a missing or insufficiently strong annotation (loop invariant, assertion, etc.) rather than to an error, and to write such additional annotations.

On the contrary, a counter-example immediately shows that the program does not meet its current specification, and prevents the waste of time of writing additional annotations. Moreover, the concrete test inputs and corresponding program path reported by the testing tool precisely indicate the erroneous situation and help the validation engineer to find the problem.

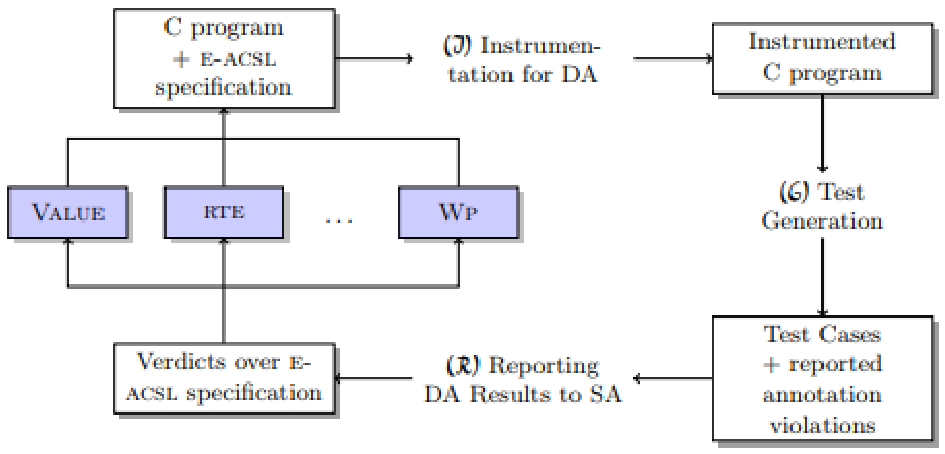


Figure 2: STADY tool process

In the figure above, (J) Instrumentation translates ACSL annotations into executable C code, (G) is the test generation with PathCrawler, and (R) is the reporting and interpretation phase of the results.

2.4 Optimizing testing by value analysis and weakest precondition (LTest)

LTest tool is a recent tool, presented in [LTEST] [7].

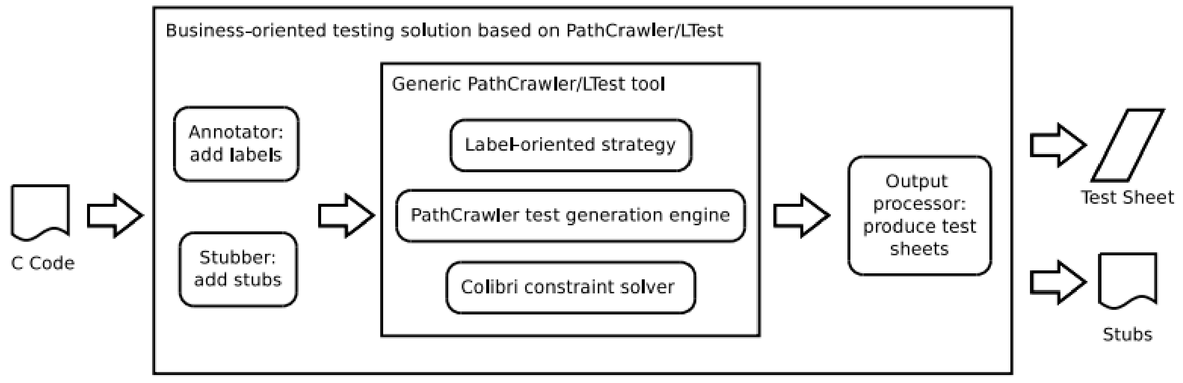
In current software engineering practice, testing is the primary approach to find errors in a program. Testing all possible program inputs being intractable in practice, the software testing community has long worked on the question of test selection: which test inputs to choose in order to be confident that most, if not all, errors have been found by the tests. This work has resulted in proposals of various testing criteria, including code-coverage criteria. A coverage criterion specifies a set of test requirements or test objectives, which should be fulfilled by the test suite (i.e., the set of test-cases). Typical requirements include for example covering all statements (statement coverage) or all branches (decision coverage) in the source or compiled code.

Code coverage criteria are widely used in industry. In regulated domains such as aeronautics, code coverage criteria are strict normative requirements that the tester must satisfy before delivering the software. In other domains, they are recognized as good practice for testing.

However, automatic tools for the generation of test inputs to satisfy code coverage criteria have not yet made it into widespread industrial use, despite the maturity of the underlying technology and the promise of significant gains in time, manpower and accuracy. This reticence is probably cultural in part: an automated test process can be very different to a manual one and test engineers who are used to functional testing have to accept the idea that an automatic tool can generate test inputs to respect a code coverage criterion but cannot provide the oracle. It can no doubt also be explained by the very importance of the test process: businesses may be reluctant to conduct experiments in such a crucial part of the development cycle. Finally, existing test tools might not correspond closely enough to the needs of industrial users and cannot easily be integrated into existing processes [7].

This is the gap which has to be closed in order for automatic structural testing tools to be used in an industrial setting and LTEST is the implementation of such a tool to be integrated into industrial practice.

Figure 3: LTest tool process


Business-oriented testing solution based on PathCrawler/LTest
Generic PathCrawler/LTest tool
Annotator: add labels
Stubber: add stubs
Label-oriented strategy
PathCrawler test generation engine
Colibri constraint solver
Output processor: produce test sheets
C Code
Test Sheet
Stubs

The generic test generation tool PATHCRAWLER/LTEST provided by the CEA contains three main ingredients (see figure above). A concolic testing tool, CEA's PATHCRAWLER, is used to generate test-cases for a given C program. The generation of concrete test inputs for a given program path relies on a constraint solver, COLIBRI. The specification mechanism of labels and a specific label-oriented strategy allow an efficient support of a desired test coverage criterion expressed as labels.

To adapt PATHCRAWLER/LTEST for a specific industrial context, additional modules were developed by CEA. They include ANNOTATOR (that expresses the specific target criterion in terms of labels), STUBBER (that produces necessary stubs) and OUTPUT PROCESSOR (that creates the required test reports).

LTest exposed awesome results during a first experiment in an industrial context [7]. It definetely represents a successful approach to code coverage.

2.5 Accelerating runtime assertion checking by static analysis (E-ACSL)

E-ACSL [1] is a plug-in distributed with the Frama-C platform. E-ACSL automatically translates an annotated C program into another program that fails at runtime if an annotation is violated. If no annotation is violated, the behaviour of the new program is exactly the same – excepting potential performance and concurrency issues – as the one of the original program (a simple example is given in the next chapter dedicated to the CURSOR method). In recent publications [3] and [4], CEA elaborates on how static analysis is exploited in E-ACSL, in order to optimize the instrumented code generation.

E-ACSL translation brings several benefits. First, it allows a user to monitor C code and perform what is usually referred to as "runtime assertion checking". This is the primary goal of E-ACSL. Indirectly, in combination with the (WP) RTE (RunTime Error Frama-C plug-in), this usage allows the user to detect undefined behaviors in its C code. Second, it allows combining Frama-C and its existing analyzers with other C analyzers that do not natively understand the ACSL specification language. Third, the possibility to detect annotations with *Invalid* status during a concrete execution may be very helpful while writing a correct specification of a given program, e.g. for later program proving. Finally, an executable specification makes it possible to check assertions that cannot be verified statically and thus to establish a link between runtime monitoring and static analysis tools such as EVA or WP.

Annotations used by the plug-in must be written in the E-ACSL specification language, a subset of ACSL. E-ACSL plug-in is still in a preliminary state: some parts of the E-ACSL specification language are not yet supported.

In Annex 1, we also present an innovative approach developed in the project, published in [2], and partly based on E-ACSL. It introduces a new notion of verification coverage that permits to combine test and proof for verifying a group of specifications related to the same piece of code, subsuming both the usual notions of functional coverage and structural coverage. A new algorithm that automatizes most parts of a verification campaign combining test and proof in order to complete the verification process as quickly as possible, is also exposed in details.

2.6 Static and dynamic checking for security counter-measures (CURSOR)

CURSOR method was initially designed at the very end of the FP7/STANCE project (2012-2016). In the context of VESSEDIA, this method is assessed on real use cases (in WP5), improved in terms of performance and accuracy in particular by means of the upcoming releases of the Frama-C E-ACSL and EVA (formerly Value) plug-ins, and enlarged to pentesting/fuzzing approach (also in WP5, planned until M36) by combinations with the well-known AFL (American Fuzzy Lop) fuzzer [11]. At first glance, CURSOR is an “all purpose” approach and method, but it may also include its own complementary analysis tools and techniques to be able to address specific case studies.

In the following, the original definition of CURSOR is presented. In brief, it combines static analysis (common weaknesses identified in the source code), translation of these weaknesses as executable statements, and finally testing (which can involve widespread fuzzing techniques, in particular the AFL fuzzer which combines with genetic algorithms to optimize efficient input test case discovery and code coverage).

General purpose

The methodology named CURSOR and proposed in this section involves some of the tools presented earlier in this report, experimented and applied to DA’s use case in WP5. This approach 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 components of interest rely on CVEs or exploits, 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 was one of the main purposes of FP7/STANCE project, with extensions and improvements planned in H2020/VESSEDIA.

On one hand, a sound static analyzer, like 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[8] 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 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 [6], by generating counter-examples when

[8] Annotations that will need to be discharged later by other complementary means.

applicable. These counter-examples can also be obtained by different means, from random testing to constraint solving techniques or even fuzz testing. Penetration testing tools as such provide also dynamic analyses generally on binary files and larger targets of evaluation. But once again, there is no guarantee of exhaustiveness, no guarantee that all vulnerabilities in the source code will be duly detected.

In brief, 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."* [9]

The output of each step in this method is a source code with potentially some annotations added by the analyzers. For convenience, only the most significant steps are presented in the following, thus hiding some implementation workarounds which present no intrinsic interest for this guideline.

Basically, the method consists in **generating automatically as many ACSL alarms** as the static analyzers will find (like Frama-C EVA plug-in), then **discharging some of these alarms** by means of static (like Frama-C WP[10]) and dynamic analyses, and finally **translating with E-ACSL** the remaining annotations as executable code for further pentesting/dynamic analyses and eventually operations. For the dynamic analysis purpose, AFL (American Fuzzy Lop) fuzzer[11] will be explored in WP5.

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 analyzers. This represents an important improvement compared to other approaches only based on the (potentially costly) static proof of the

[9] In this methodology, we intend to extend the Runtime Assertion Checking to operational executions (and not only testing/simulation phases).

[10] As using WP can be costly in terms of expertise and time, its use will be restricted to some small sub-parts of the source code.

[11] https://lcamtuf.coredump.cx/afl/

absence of the targeted CWE in the source code, or uncomplete testing of the possible application input domains.

A multi-step process

The figure hereafter illustrates the different steps of the methodology proposed in this guideline. Its breakdown is as follows:

- Preliminary studies on the use case

Some tools are of interest for a first overview of the code under verification, or to analyze more in depth some control or data-flows that may be responsible for vulnerable behaviours of the application under security evaluation.

Thus the tools required are: the EVA/Value plug-in (in simple "push-button" mode for preliminary analysis), and possibly the Slicer to reduce the code to some interesting sub-parts based on criteria like the alarms raised by EVA.

This list of tools is in no way exhaustive of course. Other Frama-C plug-ins (RTE, etc.) or any other analysis, tooled or not, may be used to get any relevant information from the source code under study.

- Frama-C EVA (formerly Value plug-in)

EVA analysis is designed to raise alarms on statements that could lead to potential common runtime errors (overflows, unbound arrays, etc.): in that case, ACSL annotations are added to the code, just before the statement (i.e. the statement could be considered for further analysis only if the annotation is valid).

For a given library, the source code can be analyzed by EVA, if necessary function by function, thanks to a script ad hoc: 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 by DA, enlarges the set of CWE categories caught by EVA. As a reminder, 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 related to taint analysis and for instance double release/close of the same resource. These new alarms are mostly 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 stored as ACSL annotations. These alarms are related to common CWEs empirically involved in most of the CVEs observed in service history.

- Dynamic approaches

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), and thus limiting the additional code that will be generated later by E-ACSL. They generally require no expertise, especially in formal methods.

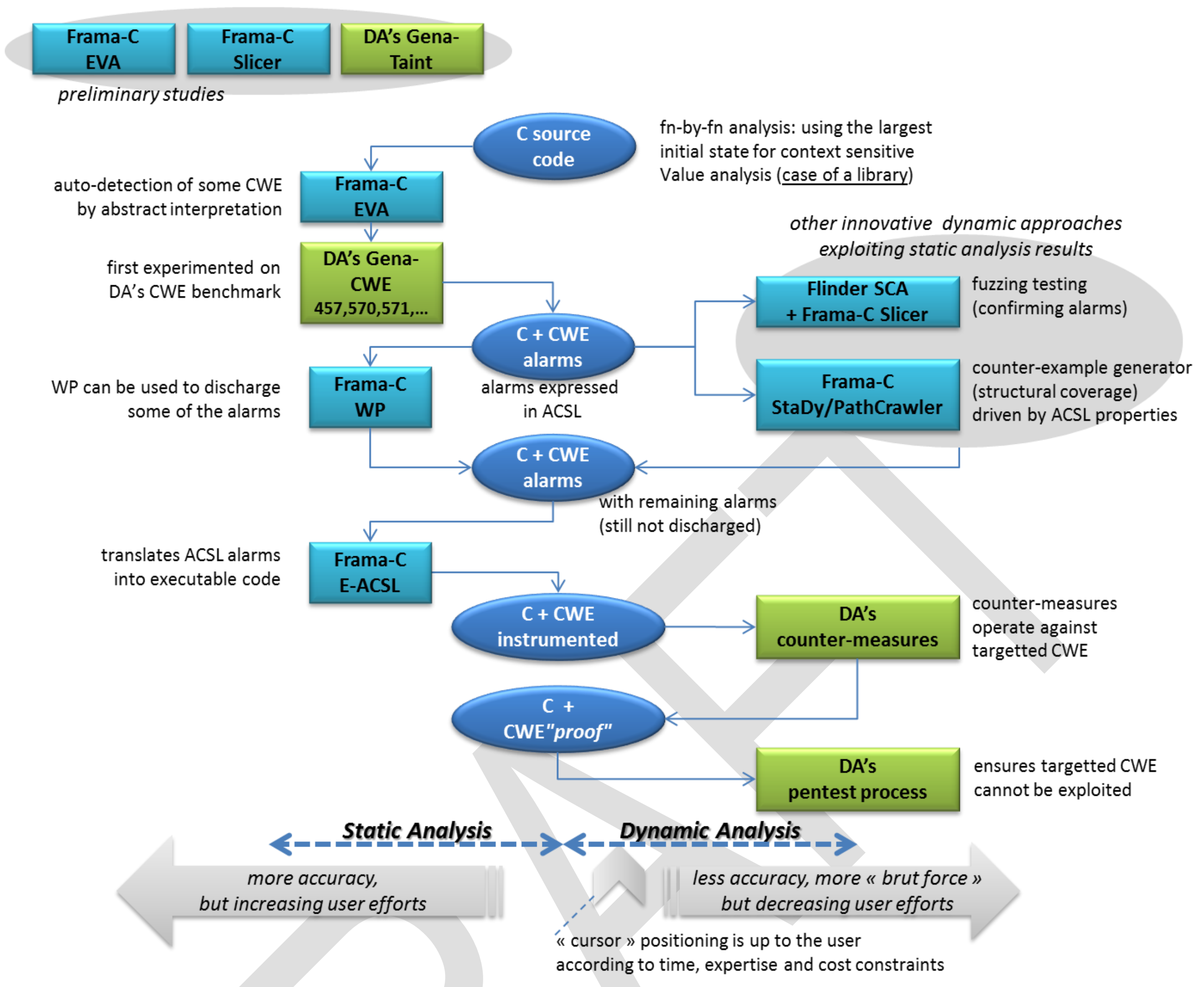

Frama-C EVA
Frama-C Slicer
DA's Gena-Taint
preliminary studies
C source code
fn-by-fn analysis: using the largest initial state for context sensitive Value analysis (case of a library)
auto-detection of some CWE by abstract interpretation
Frama-C EVA
first experimented on DA's CWE benchmark
DA's Gena-CWE 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
DA's counter-measures
counter-measures operate against targetted CWE
C + CWE"proof"
DA's 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 4: CURSOR: Methodology overview
(*original FP7/STANCE project version*)

- Frama-C WP

WP, as presented earlier, 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 (the Annex 2 to this guideline will clearly illustrate this idea). 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, as expressed previously, this task should be limited to restricted 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. As noted previously, in its current release, 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 Gena-CWE, ...), and for the scope and security objectives targeted by our use case in the context of STANCE.

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 (formerly RTE or CWE alarms). This guideline is not intended to present these counter-measures. Let us simply list the range of behaviours they may represent: from logging, 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 programming.

Then, the source code obtained at this step can be considered in theory 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.

It is also possible to make use of a fuzzer to discharge or confirm alarms, as it will be illustrated in WP5.

A simplified process

A simpler process can also be defined, in which the final user is not willing to put any effort on static analysis tools.

Therefore, we propose below a simplified verification and counter-measure process in which only automatic static analyzers are involved, in a full "push-button" approach. Several steps from the previous process are thus removed. The process is then quite straightforward as presented in the figure below: it consists in analyzing the code by means of EVA and Gena-CWE 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.

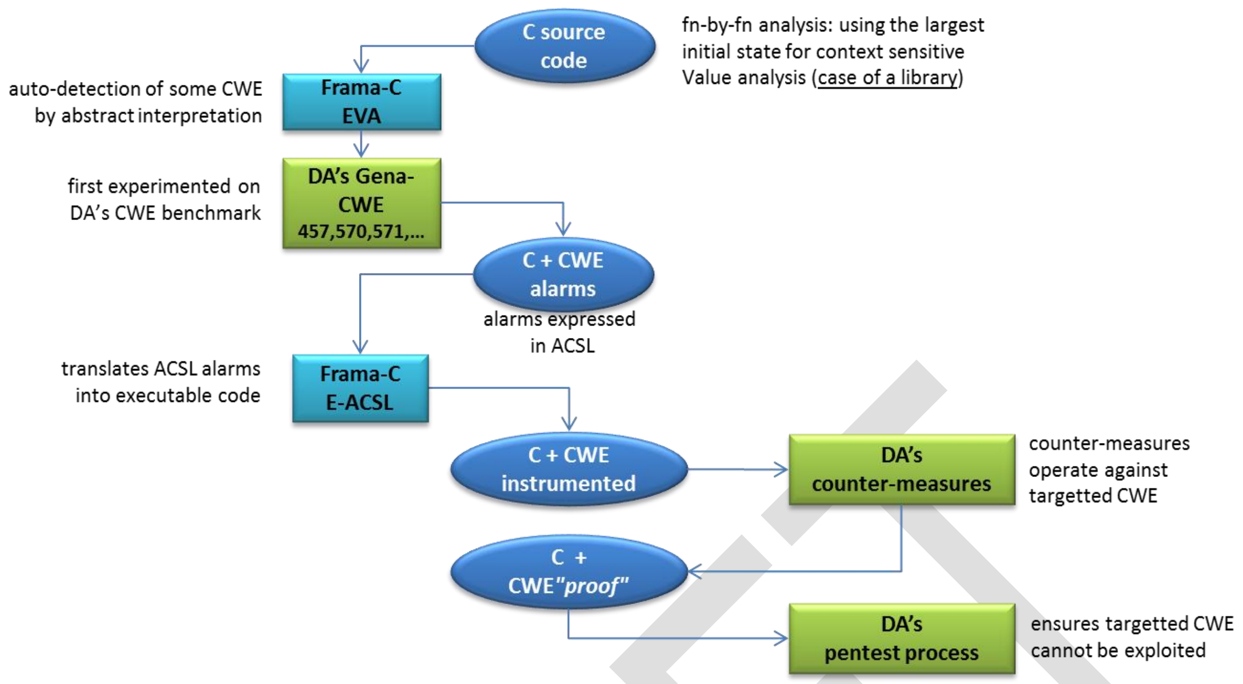

C source code
fn-by-fn analysis: using the largest initial state for context sensitive Value analysis (case of a library)
auto-detection of some CWE by abstract interpretation
Frama-C EVA
first experimented on DA's CWE benchmark
DA's Gena-CWE 457,570,571,...
C + CWE alarms
alarms expressed in ACSL
translates ACSL alarms into executable code
Frama-C E-ACSL
C + CWE instrumented
DA's counter-measures
counter-measures operate against targetted CWE
C + CWE"proof"
DA's pentest process
ensures targetted CWE cannot be exploited

Figure 5: CURSOR: Simplified methodology
(*original FP7/STANCE project version*)

In VESSEDIA WP5 (task 5.3, at M36), this method coupling static and dynamic techniques will be duly extended and improved, by involving widespread fuzzing solutions.

2.7 Static and dynamic checking of information flows (SecureFlow)

SecureFlow Frama-C plug-in is presented in [10].

Information flow analysis models the propagation of data through a software system. It identifies unintended information leaks to guarantee confidentiality of information. For instance, secret data are often forbidden from influencing public outputs. This is an instance of the non-interference property which states that certain kinds of computations have no effects on others. A more precise standard property is termination-insensitive non-interference (TINI), i. e., non-interference without taking into account covert channels due to termination.

Information flow analyses for ensuring TINI can be static or dynamic. The former examines the source code without executing it, while the latter checks the desired properties at runtime. Dynamic monitors have neither knowledge of commands in non-executed control flow paths, nor knowledge of commands ahead of their execution. Such dynamic monitors cannot be sound with respect to non-interference, while being at least as permissive as flow-sensitive type system.

Flow-sensitivity means that the same variable may carry data of different security levels (e.g. secret and public) over the course of the program execution. It is particularly important in practice in order to accept more programs without jeopardizing security. This leads to hybrid information flow in which the dynamic monitors are helped with statically-computed information.

TINI monitoring was refined by introducing the concept of termination-aware non-interference (TANI). TANI monitors are required to enforce TINI with the additional constraint that they do not introduce new termination channels. That is, no information about secret values may be derived from the fact that the monitored program terminates normally. The authors prove that hybrid monitors such as ours do enforce this property. Flows tracked by monitors may be either explicit or implicit. Explicit flows are propagated through assignments when assigning secret data to a memory location which therefore becomes secret. Indirect flows are usually propagated through program control dependencies. For instance, when considering a sensitive variable secret and the code snippet

```
if (secret) x = 0; else y = 1;
```

both variables x and y become sensitive because the fact whether secret is zero leaks to the values of both x and y. In order to detect such a flow at runtime when executing the then-branch (resp. else-branch), it is required to have the knowledge of the update of y in the non-executing else-branch (resp. x in the then-branch). This information is unfortunately not available at runtime: a static analysis using points-to information is necessary to pre-compute it.

A hybrid analysis is proposed through a sound program transformation which encodes the information flows in an inline monitor to detect TINI (and TANI) violations. A prototype is implemented as a Frama-C plugin named SecureFlow. The soundness of the program transformation relies on a static analysis in order to compute over-approximations of written memory locations in some pieces of code (also in the presence of arrays, complex structures). One benefit of a transformation-based approach is that it lets end-users choose their verification techniques: they can verify all execution paths of the generated program by static analysis, or some individual paths by runtime dynamic verification, or even use a combination of both.

The tool was assessed on an open-source cryptographic library by combining static and dynamic techniques [10].

Chapter 3 Considerations on methodology

Due to the testing and verification broad domains, and the specificities of development and V&V in software industry, providing the reader with a complete and universally applicable methodology could not be possible in the scope of this task and guideline. Indeed, most of the couplings presented in Chapter 2 already contain elements of methodology which should clarify if they are applicable to some use case of interest under study.

However, some considerations may be of some interest for new comers in formal analysis. Several ‘helper’ plug-ins provided within the Frama-C platform could help the verification analysis and / or testing, whatever the V&V process step and for a wide category of code and properties (if expressible in ACSL).

Typically, the Slicer can be widely used during or even before the analyses, to reduce the code with regards to either given annotations or control points in the code. Slicing code and annotations permits to focus on smaller pieces of program, execute it more rapidly, reduce the size and/or the number of proofs of obligation to be discharged, and so on. In other words, the Slicer will permit to decompose a given problem into smaller pieces. Of course, it is up to the specialist to ensure or justify that the verification of the slices is somehow equivalent to the verification of the whole program. Another subtlety is that if the code to slice is too big or if the EVA analyser is not able to be accurate enough (i.e. when its computations yield too large domains of value or even more non-conclusive over-approximations), then the code sliced could be as heavy to analyse as the original code. In other terms, the slicer might not be of any help in all circumstances. This has to be consolidated, given the fact that only experimentations will conclude.

An alternative dynamic/static method

The document in Annex 2 (a FOKUS’ experimentation on testing and verification) provides another approach and illustrations for positioning testing and formal analysis, in particular with the Frama-C toolbox, performed in the scope of VESSEDIA. These considerations are illustrated with a C++ library function code: *unique_copy*. It appears that in most cases both approaches would need a certain amount of efforts to validate any requirements of interest. It concludes that “*the process of finding the necessary code annotations involves a lot of guessing whose results are best checked with some test data before one ventures to prove them*”.

Chapter 4 Application to WP5 Use Cases

For the Contiki OS, we will make use of WP, EVA and E-ACSL. Some low level core components of Contiki have already been verified (including functional properties) using the WP plugin of Frama-C. However, for client code using those features, it may be too difficult or simply unnecessary to perform such a detailed verification with WP. Thus, we conducted some experiments to make the ACSL specification of low-level modules executable to check at runtime, with E-ACSL, that the contracts of the APIs are respected. Another aspect is the verification of absence of runtime errors. We will combine the use of Frama-C/EVA and Frama-C/E-ACSL. Basically, we will first execute EVA on each test of Contiki to generate the potential failing assertions and then use E-ACSL to execute the corresponding test for the native platform (x86). Indeed, for other platforms, there is no E-ACSL library and the limited computing power (and memory) of the platform would probably make impossible an efficient port of this library since, for example, it is already hard to compile some tests for their target platform due to the memory limitation, even without any instrumentation.

For the code analysis of the 6LowPAN management platform use case, the use of STADY tool will be investigated to perform test generation during the Frama-C WP-based verification of the source code associated to both MPL routing (for firmware transfer in the 6LowPAN network) and the read/write operations on the firmware in the 6LowPAN node’s Flash memory. The objective behind this work is to identify any counterexample to assess the strength of a number of written annotations like loop invariants.

For the experimental AMS use case, we will apply E-ACSL, EVA and a fuzzer (AFL) coupling, following the CURSOR method guideline. This static and dynamic analysis combination was first experimented earlier in the project (in the scope of WP5). This approach is expected to be straightforward, say “push-button”, as it does not intend to discharge all of the alarms (in particular spurious ones!) found by static analysis, and thus does not require specific expertise.

Chapter 5 Summary and Conclusion

This report presents a guideline for combining, coupling static and dynamic approaches, contributing to safety and security of pieces of program when the source code is available.

The tools taken into account here are the ones provided, and for some of them improved, in the scope of VESSEDIA project.

Most of these tools and approaches illustrate the capability of static tools to be extended and/or interfaced with dynamic functionalities. This is of high interest as usually V&V teams in industry are used to dynamic verification, thus testing coverage. Testing provides of course an interesting level of confidence. However, testing is by nature not exhaustive, thus the testing coverage has to be improved. Hence, static approaches can be of important help: increasing the coverage level, focusing on "first rank" alarms, taking advantage of innovative methods from A.I. like genetic algorithms to optimize the search for input data example and counter-examples, etc.

Thus, this guideline aimed at encouraging dynamic "testing" practices to progress towards static analyses while retaining (say, marginally changing) the good classical practices in industry. The expected benefits rely on static analysis providing better confidence on applications: static tools work on whole sets of behaviours, and not only sampled inputs and states.

Some first insights are also provided regarding application to VESSEDIA use cases. The results will mostly be presented in the scope of WP5. First attempts, however, showed promising preliminary results, in line with the expectations seen in this document. As combining static and dynamic approaches remains an intensive field of explorations, we definitely recommend them as a first step, or bridging philosophy, between classical dynamic verification activities, and more formal static experimentations.

Chapter 6 List of Abbreviations

Abbreviation	Translation
ACSL	ANSI/ISO C Specification Language
EVA	Evolved Value Analysis
V&V	Verification and Validation

Chapter 7 Bibliography

[1] E-ACSL Plug-in (Release Chlorine-20180501). Julien Signoles and Kostyantyn Vorobyov. https://frama-c.com/download/e-acsl/e-acsl-manual_Chlorine-20180501.pdf

[2] Viet Hoang Le, Loïc Correnson, Julien Signoles, and Virginie Wiels. Verification Coverage for Combining Test and Proof. In International Conference on Tests and Proofs (TAP'18), June 2018.

[3] Fonenantsoa Maurica, David R. Cok, and Julien Signoles. Runtime assertion checking and static verification: Collaborative partners. In International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2018), November 2018.

[4] Dara Ly, Nikolai Kosmatov, Frédéric Loulergue, and Julien Signoles. Soundness of a dataflow analysis for memory monitoring. In Workshop on Languages and Tools for Ensuring Cyber-Resilience in Critical Software-Intensive Systems (HILT 2018), November 2018.

[5] Combining Static and Dynamic Analysis for Bug Detection and Program Understanding, Kaituo Li (2016). University of Massachusetts Amherst Doctoral Dissertation. https://scholarworks.umass.edu/cgi/viewcontent.cgi?article=1845&context=dissertations_2

[6] Kiss B., Kosmatov N., Pariente D., Puccetti A. (2015) Combining Static and Dynamic Analyses for Vulnerability Detection: Illustration on Heartbleed. In: Piterman N. (eds) Hardware and Software: Verification and Testing. Lecture Notes in Computer Science, vol 9434. Springer, Cham.

[7] Test Case Generation with PATHCRAWLER/LTEST: How to Automate an Industrial Testing Process. Sébastien Bardin, Nikolai Kosmatov, Bruno Marre, David Mentré, and Nicky Williams. In Proc. of the 9th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISOLA 2018), Limassol, Cyprus, November 2018. http://nikolai.kosmatov.free.fr/publications/bardin_kmmw_isola_2018.pdf

[8] Combining Static and Dynamic Analysis for Vulnerability Detection. Sanjay Rawat, Dumitru Ceara, Laurent Mounier, Marie-Laure Potet (2013). https://arxiv.org/pdf/1305.3883

[9] The SANTE Tool: Value Analysis, Program Slicing and Test Generation for C Program Debugging. Omar Chebaro, Nikolai Kosmatov, Alain Giorgetti, Jacques Julliand (2011). https://hal.inria.fr/inria-00622904/document

[10] Hybrid Information Flow Analysis for Real-World C Code. Gergö Barany and Julien Signoles. In International Conference on Tests and Proofs (TAP'17), July 2017.

[11] Static Analysis and Runtime-Assertion Checking: Contribution to Security Counter-Measures. Dillon Pariente, Julien Signoles, Symposium sur la sécurité des technologies de l'information et des communications, SSTIC'2017 - Rennes, France June 7-9 2017. https://www.sstic.org/media/SSTIC2017/SSTIC-actes/static_analysis_and_runtime-assertion_checking_con/SSTIC2017-Article-static_analysis_and_runtime-assertion_checking_contribution_to_security_counter-measures-pariente_eiaEnuw.pdf

[12] StaDy: Deep Integration of Static and Dynamic Analysis in Frama-C. Guillaume Petiot, Nikolai Kosmatov, Alain Giorgetti, Jacques Julliand (2014). https://hal.inria.fr/hal-00992159/document

Annex 1

Annex 2