PREMIÈRE MINISTRE

Liberté
Égalité
Fraternité

Agence nationale de la sécurité
des systèmes d'information

Secrétariat général de la défense
et de la sécurité nationale

Paris, le 27 septembre 2022

Référence : ANSSI-CC-NOTE-26_v1.0

APPLICATION NOTE

Requirements for tool-based static code analysis in security evaluations

Applicability : On approbation.

Confidentiality : Public.

SÉCURITÉ CERTIFICATION

VERSION HISTORY

Edition	Date	Modifications
1.0	31 may 2022	First version

TABLE OF CONTENTS

1 Scope and applicability

This note concerns CC evaluations requiring compliance to requirements AVA_VAN.3 or higher.

In this note, the notion of 'source code' is restricted to source written in programming languages, whether Compiled or interpreted (*e.g.* C, Java, Python or PHP), as opposed to hardware description languages (e.g. VHDL or Verilog), which are considered out of scope.

1.1 Motivation

ANSSI observed that the [CEM] lacked details on the code analysis activities required to correctly assess this compliance. This note aims at closing this gap, albeit only for software code analysis (i.e. lower-level code, such as hardware description language, is not considered).

EXAMPLE: let us assume that a connected device is submitted to an AVA_VAN.3 evaluation, and includes:

- a bootloader written in ASM,
- an FPGA used for cryptographic functions, and
- a firmware written in C.

This note will apply only to the code review of the firmware.

1.2 Role of implementation in vulnerability analysis

The motivation of this note originates from the AVA_VAN.3.3E/AVA_VAN.4.3E/AVA_VAN.5.3E requirements. These requirements state that the evaluator performs *a vulnerability analysis of the TOE using the guidance documentation, functional specification, TOE design, security architecture description and implementation representation to identify potential vulnerabilities in the TOE*.

The implementation is needed at that stage to identify two different types of potential vulnerabilities:

- some logical vulnerabilities, generic to the product type, that can only be checked at the implementation level. This activity is out of scope of this note, and should be performed according to the [CEM]

  EXAMPLE:

  - in a smartcard, lack of countermeasures against fault injections,
  - in an application, presence of residual information such as hardcoded passwords in the code;

- vulnerabilities introduced by a bad use, or a limitation, of the implementation technology itself (programming language, compiler..);

  EXAMPLE: In a C program,

  - a buffer overflow caused by the use of strcpy,
  - a secret key left in memory because its zeroisation was skipped by the compiler (due to optimization options).

This note addresses only the vulnerabilities introduced by a bad use, or a limitation, of the implementation technology itself. The main requirement of this note comes from the return of experience of the French certification body, with regard to manual[1] static analysis and dynamic

[1] e.g. a critical reading of the source code

analysis[2][3] (automated or not). It was observed that as a general rule, manual static analysis and dynamic analysis are always likely to miss significant vulnerabilities, even when reviewing small codebases.

Consequently, it has been considered necessary to mandate the use *automated static analysis* in order to demonstrate compliance in a manner both complete and comparable between ITSEFs. Therefore ANSSI considers this requirement as the most direct interpretation of [CEM] in today's state-of-the-art.

It should be noted that during an evaluation, manual analysis may be used for other purposes, such as traceability of SFRs or as a complement to functional tests. Those activities and out of scope of this note as well. More generally, the note has no impact on activities performed in the context of other requirement families than AVA_VAN and ADV_IMP.

[2] i.e. implying the execution of the interpreted or compiled program

[3] dynamic analysis requires the execution of the compiled or interpreted program: for instance, functional tests or fuzzing.

2 Requirements for the sponsor

E1. The sponsor shall provide the full source code of the TOE to the evaluator and identify all the parts of the TOE that are implemented by closed-source COTS.

Remark: this requirement is directly taken from [CC][4] for ADV_IMP.1: even if the evaluator does not analyse the full source code, they must still be able to define independently and impartially what sample they want to analyse. Moreover, the requirement allows the evaluator to correctly rebuild the TSF. Evaluators may reject incomplete source code by referring to section 3.5.4 of [NOTE-20].

Remark: open source components of the TOE are not mandatorily delivered to the ITSEF, as long as the sponsor provides clear directives allowing the ITSEF to recover them (exact version, link to a *github* project, and so on) and a written statement confirming that they have not modified these components.

Remark: the certification body may allow a sponsor to provide access to source code on their premises instead of delivering it to the ITSEF. This may only be done on a case-by-case basis, and will require assurance that the analysis by the ITSEF is not hampered in any way (the ITSEF should be able, for example, to install their tools in the developer premises). This however is not allowed for evaluations including AVA_VAN.4 and higher.

Remark: by definition, the code analysis cannot be performed on closed-source COTS. It is therefore crucial to identify all such components.

E2. The sponsor shall provide the evaluator with all necessary elements required to build the TOE (tools, scripts…).

Remark: this requirement is directly taken from [CC] and allows completeness[5] as well as ensures the good understanding of the implementation by the evaluator[6]

Remark: this requirement implies that proprietary tools must be delivered as well, when they are required to build the TOE. "Building" the TOE has to be understood in the software sense.

EXAMPLE: for an embedded device like a Smart Meter Gateway, "building" would mean "compiling the firmware part(s) that can be loaded on the TOE itself".

Warning: if the sponsor has to provide the tools, they are not mandated to document them. (this is only required if ALC_TAT.1 is selected). However, the sponsor must assist the evaluator in building the TOE – this support is expected following [NOTE-20]. In the same spirit, if the analysis requires code stubbing[7], the sponsor must assist the evaluator so as to ensure that the stubbing is made in a relevant fashion.

[4] Cf. Part 3 §13.3 "The entire implementation representation is made available to ensure that analysis activities are not curtailed due to lack of information. This does not, however, imply that all of the representation is examined when the analysis activities are being performed."

[5] Cf. Part 3 ADV_IMP.1.1C "The implementation representation shall define the TSF to a level of detail such that the TSF can be generated without further design decisions"

[6] Cf. Part 3 §13.3 "it is important that such "extra" information or related tools (scripts, compilers, etc.) be provided so that the implementation representation can be accurately determined"

7 A stub is a temporary alternative to a code that cannot be used directly by another code. This alternative should ideally have the same functional characteristics than the replaced code – if not possible, it can be a code that does not implement any action, and returns always the same result, or a simplified version of the initial code. This process aims at allowing the static analysis to go on, even if some parts of the code cannot be analyzed directly by the tool.

3 Requirements for the evaluator

E3. If AVA_VAN.3 (or higher) is selected, the evaluator must define and use a tool-based static code analysis methodology that meets the table of contents given in Annex A. Notably, the methodology must be based on code analysis tools, following a structure approach to vulnerabilities. Manual analysis is to be used only as a complement to tool-based analysis, in particular when interpreting the outputs from the tool, define the criticality of the findings, and perform checks that the tool is not able to perform[8].

Remark: [CEM] does not require the use of tools: it is a requirement by the certification body due to the state-of-the-art.

Remark: the presentation of the methodology is not fixed. The ITSEF should define the appropriate presentation so that evaluators are properly trained and skilled in practice. For example, the method can take the form of a training support or inline help for analysis tools.

Remark: the method will be validated as part of the ITSEF licensing process.

E4. If AVA_VAN.5 is selected, the code analysis must cover the whole source code.

Remark: this requirement comes from the fact that AVA_VAN.5 includes attacks up to 24 points according to [CC], which include exhaustive analysis of the source code:

a) *Elapsed Time: 1 month (4);*
b) *Expertise: on software expert (7);*
c) *Knowledge of TOE: knowledge of the entirety of the source code, including the most critical parts (11);*
d) *Window of Opportunity: attacks that do not require a specific WoO (0);*
e) *Equipment: specialized SW (2, as per* [NOTE-18]*).*

E5. If AVA_VAN.3 or AVA_VAN.4 is selected, the evaluator shall give an attack-driven rationale for their sampling.

Remark: this implies that the evaluator cannot simply select the parts of the source code that implement the security functions. Indeed, parts that are « SFR non interfering » can include vulnerabilities allowing arbitrary code execution of the TOE. Quite the contrary, it is recommended to focus on the code easily accessible to the attacker (e.g. TSFI) and then add other parts following the collected evidence.

Remark: as a general rule, *open source components used in the TOE must be part of the analysis. However, as the entry point of the analysis is in the proprietary code, the tool will analyse only the code that is actually used by the TOE.*

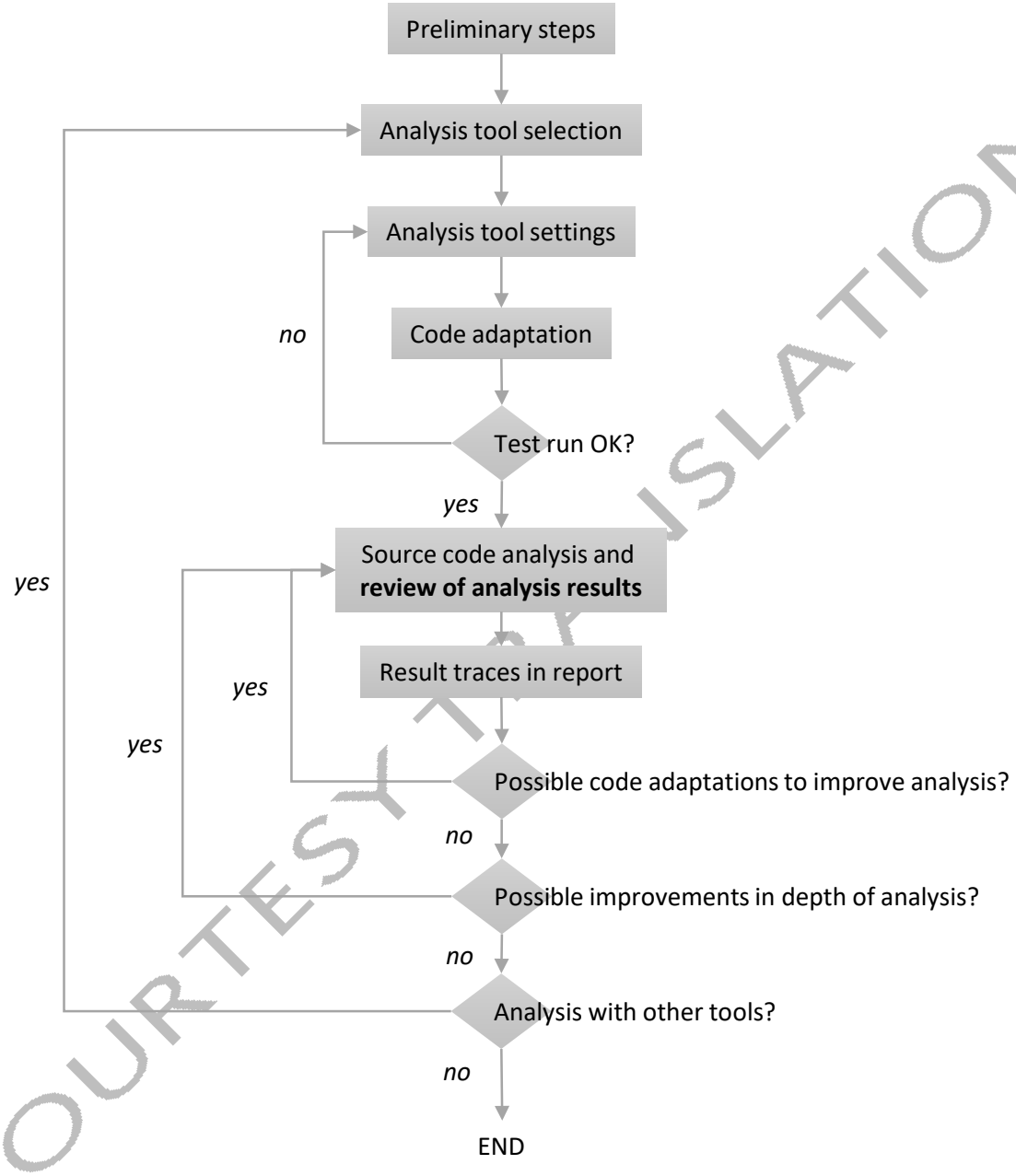
[8] E.g.:

- Check the implementation of an algorithm or protocol (RFC, crypto spec….),
- Check the presence/absence of a vulnerability, the semantics of which is not detectable by the tool (e.g. masking of confidential data…),
- ….

ANNEXE A. Table of contents for code analysis methods

1. Introduction

The main sections of this method correspond to the steps summarized in the figure hereafter. All steps are mandatory.



2. Preliminary steps

2.1. Inputs from the developer

The methodology shall describe how the evaluator recovers and manages the necessary inputs:

- *All source files of the TOE;*
- *All makefiles/scripts (or other project management files);*
- *Third party components (such as libraries).*

2.2. Analysis of the compilation

The same code gives different runtime behaviours depending on the compilation options: therefore, the evaluator must examine the code and the compilation process, so that their analysis is complete. Compilation options must also be considered while studying the exploitability of code vulnerabilities.

The methodology shall describe the following activities:

- *Clear identification of the tools used for the build:*
  - *Reference and version of the building tools (such as compilers),*
  - *Presence of proprietary tools (in that case the evaluator is mandated to ask the developer to provide the tool),*
  - *Hardware target(s);*
- *Verification that the compilation step is successful and can be repeated independently by the evaluator at their premises;*
- *Analysis of the compilation phase – typically check the makefiles to ascertain the use of:*
  - *Good practices,*

    EXAMPLE: some options can allow developers to detect errors at the compilation phase, such as the gcc option -Wall -Wextra. See also e.g. sections 5.1 et 5.2 of [Guide C]
  - *Hardening.*

    EXAMPLE: some options can help mitigate the exploitation of errors at runtime, such as the gcc option -Wformat. See also e.g. section 5.3 of [Guide C].

This phase may include back-and-forth exchanges between evaluator and developer: for large codebases, the evaluator may require that the delivered code already passes successfully a compilation with a restrictive set of options, so as to reduce the potential number of errors during the static analysis.

The methodology shall require the evaluator to provide the exact configuration for the build (e.g. complete list of options)

3. Analysis tool selection

3.1. List of static analysis tools used by the evaluator

The methodology shall select tools that detect code vulnerabilities (e.g. based on security coding guidelines as CERT, ISO 17961 or ANSSI guidelines, and on runtime-error detection).

NOTE:

- *Extended grep-like tools can be used but are not sufficient;*
- *Analysis tool oriented towards code quality and metrics are also welcome but do not address the needs of this methodology.*
- *The selected tool must have a code coverage feature to identify potential misuses*[9] *of the tool and to know precisely which parts of the code were analysed.*

[9] An insufficient code coverage is a good indicator of a bad configuration of the tool.

3.2. Identification of constraints

The methodology shall define control points to check that the tool is appropriate with regard to the properties of the TOE, including at least the following:

- *The languages analysed by the tool cover the actual languages of the TOE (multi-language, possible presence of assembly code in the TOE…);*
- *The tool supports the hardware architecture of the TOE;*
- *The types of defects checked by the tool are consistent with the requirements of the evaluation (requirements may be specific to a TOE, but the methodology should select a set of requirements that will be used by default);*
- *The tool is capable to manage the code size and complexity of the TOE;*
- *The tool provides information on code coverage.*

3.3. Recommendations

As the evaluator is expected to perform an exhaustive analysis (within the defined scope) of the tool results, it is highly recommended to leverage any tool property that reduces the overhead during the analysis, notably:

- *Minimal code adaptation: The methodology should favour tools requiring minimal code adaptation.*
- *The use of a "sound" tool is only a good practice in presence of a compatible compiler/architecture/code size/code complexity: it is not mandatory;*
- *NOTE: "sound tools" are tools that can formally demonstrate the absence of a specific vulnerability. By design such tools require a high level of expertise.*
- *Level of automation: automatic ability of the tool to manage the settings via Makefile or project manager.*

4. Analysis tool settings

The methodology shall address how the evaluator is supposed to setup the selected tools, in order to ensure the relevance of the analysis. This includes, but is not limited to:

- *Identification of all the needed source files (so as to make sure that the analysis can actually take place);*
- *Support of various compilers/architectures;*
- *Possibility to define some "implementation defined" aspects for « uncommon » targets;*
- *Representation of types: size, signedness, endianness…;*
- *Definition of entry points, initialization, interrupts and so on.*

5. Code adaptation for analysis

5.1. General principles

Adaptations should only be made if necessary to use the tool, especially since the code must stay representative of the evaluated TOE. However, the evaluator should not abandon the analysis simply because adaptations were needed.

5.2. Guidance for code adaptation

The methodology shall require the evaluator to justify the adaptations made to the code, and especially demonstrate that the adaptations themselves did not introduce any flaw detected later in the analysis and did not hide other issues.

The methodology shall describe how the evaluator is supposed to adapt the code to take care of specific constructions of the compiler/architecture (e.g. "implementation-defined" code) when it is not automatically handled by the tool or by a proper configuration of the tool.

The methodology shall describe the method used by the evaluator to manage multi-language code, notably the stubbing of parts not managed by the tool (e.g. assembly code or some parts of source code with a language not handled by the tool). Stubbing is not supposed to be used to handle missing parts of the code, since the full codebase is normally accessible to the evaluator.

6. First run/test step

The first test run is meant to check whether the tool can actually manage the code.

Consequently, the method shall describe how the evaluator:

- *selects low precision settings for the tool (minimum checks);*
- *analyses the first results, i.e.:*
  - *assesses informally the code quality,*
  - *analyses the code coverage.*

If this first run shows a bad configuration (the code coverage is often a good indicator), the methodology shall mandate the evaluator to return to the previous steps (code adaptation and/or tool settings), or even return to the developer (if the code quality is lacking and requires action from the developer).

When the coverage and the code quality are acceptable with regard to the method, the evaluator can start the real code analysis.

7. Real code analyses

In order to avoid being drowned in returned defects, the methodology shall mandatorily describe a step-by-step process when the tool allows it.

- *First step: the evaluator will ultimately try to check the adherence of the code to a large set of requirements. However, in a first step, the evaluator should select a first, smaller, subset of classic flaws. As a priority, these checks should target undefined or unspecified behaviours and classic vulnerabilities.*
- *EXAMPLE: if the evaluator uses CERT-C as a set of requirements, the first run could focus on the Level 1 rules (High severity, likely, inexpensive to repair).*
- *A first review of tool results (see following section) will be performed.*
- *As a second step, the evaluator will typically increase the tool precision in a new run and perform a new review of tool results.*

EXAMPLE: if the evaluator uses CERT-C this would typically consist in adding Level 2 and Level 3 rules.

It is better, when possible by the tool, to break down the analysis in several runs, corresponding to increasingly "aggressive" settings of the tool. In all cases, a rigorous analysis of the code is expected and a single run with a low level of aggressiveness is not sufficient.

If there is remaining time after the last step of code analysis, the evaluator can activate more specific checkers in a new run to detect more complex vulnerabilities or to find issues related to code quality.

EXAMPLE: it means focusing on more complex vulnerabilities (after checking buffer overflows, the evaluator now checks TOCTOU vulnerabilities) or to look for dangerous features in the code.

8. Review of tool results

8.1. General method and interpretation guidelines

*The review of results is a **critical and non-accessory step**: tools outputs shall never be copy-pasted in an ETR without an actual expert opinion on the findings. The evaluator is expected to have the critical distance to the results provided by the tool.*

The methodology shall therefore define how this review is performed, including, but not limited to:

- *the definition of priority classes for returned defects;*
- *the method used to interpret unreachable code (initialization issue, bad settings, defensive parts of code, interrupts…), so that the evaluator is able to return to previous steps and fix the tool settings and/or adapt the code accordingly. Unreachable code can be indicative of an error in code adaptation or/and in tool settings.*

The methodology shall provide the evaluator with clear guidelines to:

- *take care of false positives (i.e. clearly identify them in the tool if possible, and keep a trace of false positives in the report);*
    - *As per [NOTE-20], the evaluator is allowed to reject the TOE if the number of findings exceeds its analysis capacity: if the evaluator considers the code as not mature enough for analysis, they may provide the list of findings to the developer and ask the developer to fix the code and/or provide a rationale that these findings are false positives;*
    - *However, the evaluator still has the final responsibility of qualify which warning are false or true positives, and will ultimately have to challenge the developer information.*
- *decide when a new tool run is needed, and with which additional checks if the remaining evaluation time allows it;*
- *decide when another tool shall be used (return to tool selection) if the remaining evaluation time allows it.*
- *The methodology shall provide guidance to rate the identified vulnerabilities according to the rating system of the evaluation method. In particular, even if the vulnerability was found by code analysis, the evaluator shall consider the possibility that the same vulnerability is found by other means. The final rating shall be the smaller rating of all considered scenarios.*
- *The methodology shall include guidelines and/or pre-ratings for the following notions:*
- *Elapsed Time: time taken to identify and exploit specific classes of vulnerabilities, e.g. by comparison to challenges or CTF exhibiting similar vulnerabilities;*
- *Level of expertise:*
    - *If the attack scenario supposes that the attacker has the knowledge of the source code, only a single expert should be required at most;*
    - *If the attack scenario supposes that the attacker can discover the vulnerability in a black box setting, additional expertise may be required (Hardware expertise to access the TOE, network expertise to perform fuzzing, etc.);*
- *Knowledge of the TOE:*

- *This rating may be low or zero in scenarios where the attacker discovers the vulnerability in a black box setting, or if the vulnerability is in an open source component;*
- *The rating should reflect the protections in place to protect the source code, which may require a developer interview: if the evaluation requires ALC_DVS.2, access to source code may be rated up to 11 points. If the developer cannot show strong access control, the rating may be as low as 3 points;*
- *The rating should also consider the possibility for an attacker to obtain a binary, and reverse engineer it to find vulnerabilities;*

- *Window of opportunity;*
- *IT hardware/software or other equipment required for exploitation: as per [NOTE-18], the rating cannot be higher than 2 for commercial tools.*
- *As a general rule, it is not recommended to try and rate all issues identified by the tool, as it would result in a very time-consuming process. Quite the contrary, the evaluator should instead try to define a general rating for a given class of vulnerabilities (for example use-after-free) and apply it to every violation of this class, unless it is a false positive.*

8.2. About the possible fixes to improve analysis

When reviewing analysis results, evaluators and developer face a practical issue: some vulnerabilities may cause the tool to stop its analysis, leaving a large part of the unanalysed code. Some tools, when finding an error or potential error, may resume the analysis by excluding the associated erroneous context(s) of this (potential) error, leaving also a part of the code unanalysed.

EXAMPLE: loop termination errors or always true/false conditions resulting in unreachable code that was not supposed to be unreachable.

The methodology shall describe:

- *in which cases the evaluator should try to fix the bugs identified in the code (or ask the developer to fix them), or when they should ask clarifications to the dev;*
- *how the evaluator is supposed to trace these cases in the ETR.*

The evaluator may simply ask for the developer to provide a fixed source code, and resume their analysis. However, other blocking errors may be found again, leading to a possibly large number of iterations between the evaluator and the developer. Alternatively, the evaluator may perform a small adaptation of the code, so that the analysis can resume. This offers the developer a more complete set of findings in a single evaluation phase. In any case, the evaluator should synchronize the further procedure with the developer before adjusting the code.

EXAMPLE: EASY/OBVIOUS FIX – use of an unsigned int counter instead of signed one resulting in an infinite loop: the evaluator changes the unsigned counter to a signed one, only to allow the analysis to resume.

EXAMPLE: the evaluator erases the whole loop that presents a termination error.

WARNING: as clearly highlighted by the previous examples, such adaptations are neither fixes nor patches, and may result in non-functional code. They are only meant to allow the analysis to resume, and should never been reintroduced in the product code as is.

ANNEXE B. References

Reference	Document
[NOTE-18]	*ANSSI-CC-NOTE-18 - Prise en compte des outils dans les évaluations logicielles*
[NOTE-20]	*ANSSI-CC-NOTE-20 - Règles relatives à la mise en œuvre des évaluations sécuritaires*
[CC]	Common Criteria for Information Technology Security Evaluation: Part 1: Introduction and general model, avril 2017, version 3.1, révision 5, référence CCMB-2017-04-001; Part 2: Security functional components, avril 2017, version 3.1, révision 5, référence CCMB-2017-04-002; Part 3: Security assurance components, avril 2017, version 3.1, révision 5, référence CCMB-2017-04-003.
[CEM]	*Common Methodology for Information Technology Security Evaluation - Evaluation methodology, April 2017,* version *3.1 révision 5,* référence CCMB-2017-04-004
[Guide C]	*ANSSI - guide - Règles de programmation pour le développement sécurisé de logiciels en langage C - v1.4 du 24/03/2022*

The documents can be found on ANSSI website (www.ssi.gouv.fr).