D1.5
Analyses choice methodology report

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

Deliverable type:	Report
Deliverable reference number:	DS-01-731453 / D1.5 / 1.0
Work package contributing to the deliverable:	WP 1
Due date:	Dec 2018 – M24
Actual submission date:	29th January, 2019

Responsible organisation:	FOKUS
Editor:	Jens Gerlach
Dissemination level:	PU
Revision:	1.0

Abstract:	This report presents the findings of Task 1.4 regarding a cost-efficient trade-off between basic and sophisticated static analyses.
Keywords:	static analysis, abstract interpretation, deductive verification, minimal contracts

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

DRAFT

Editor

Jens Gerlach (FOKUS)

Contributors (ordered according to beneficiary numbers)

Jochen Burghardt (FOKUS)
Marko Fabiunke (FOKUS)

DRAFT

Disclaimer

The information in this document is provided “as is”, and no guarantee or warranty is given that the information is fit for any particular purpose. The content of this document reflects only the author`s view – the European Commission is not responsible for any use that may be made of the information it contains. The users use the information at their sole risk and liability. This document has gone through the consortiums internal review process and is still subject to the review of the European Commission. Updates to the content may be made at a later stage.

Executive Summary

This document presents a methodology for selecting and applying static analysis methods. Based on a classification of static analysis methods into four classes (see Chapter 2) we propose a *three-level methodology as follows:*

- The first level simply leverages the capabilities of readily available resources, namely *compilers*. Their considerable capabilities to warn about suspicious or outright dangerous code fragments present a very cost-effective way to greatly reduce the number of potential vulnerabilities.
- *Heuristic static analyses* can provide more elaborate diagnostics but often come at a higher cost. We explain on what basis these methods and the related tools are to be selected and how they are best used in conjunction with the capabilities of compilers.
- Finally, we propose the use of *sound static analyses* in order to achieve a degree of assurance that heuristic analyses usually cannot provide. As in the case of heuristic analyses, sound analyses are best used after the capabilities of lower levels have been exploited.

Since we are regularly referring to the *cost* of various methods, we want to emphasise that we only do this this qualitatively. Deliverable D1.6 [1] will provide more specific cost models.

DRAFT

Contents

Chapter 1 Introduction **1**
1.1 Structure of this document 1
1.2 Background 1
1.3 Existing guidance for static analyses 2
Chapter 2 A four-level classification of static analyses **4**
2.1 Some general remarks on the combination of static analyses 4
2.2 Basic static analyses 5
2.2.1 On strong type checking 5
2.3 Simple static analyses 5
2.4 Advanced static analyses 7
2.5 Formal static analyses 8
2.5.1 Abstract interpretation 8
2.5.2 Hoare-style verification 8
2.5.3 Hoare-style verification restricted to run-time errors 9
Chapter 3 Suggestion of a three-step methodology **10**
3.1 Proper use of compiler diagnostics 10
3.1.1 Chose proper language version 10
3.1.2 Enable warning options 10
3.1.3 Discover more warning options 11
3.1.4 Enforce fixing reported warnings 11
3.2 Deployment of heuristic static analyses 12
3.2.1 Select and understand appropriate programming guidelines 12
3.2.2 Identify heuristic static analyses tools that support the selected guidelines 12
3.2.3 Enforce fixing reported problems 12
3.2.4 Reconcile with compiler diagnostics 12
3.3 Application of select sound static analyses 13
3.3.1 Understand scope of sound static analyses 13
3.3.2 Select appropriate analysis tools 13
3.3.3 Apply sound analyses after simpler analyses have been conducted 13
Chapter 4 Conclusions **14**
Chapter 5 Bibliography **15**

DRAFT

List of Figures

Figure 1:Static Analyses and Formal Methods in EN 50128 3

DRAFT

Chapter 1 Introduction

Software is written by means of programming languages which are formal languages in the sense that they have a formal grammar that can be transformed into executable code. Using a formal definition, however, does neither prevent the occurrence of software errors nor that the resulting software product is hardened against various security attacks. There are two major approaches in order to ascertain that software behaves correctly. The first approach is *testing*, also referred to as *dynamic analysis*, where the software is executed with representative inputs while the output or other observable behaviour of the software is compared with expected results. The second approach is *static analysis* where the source code or binary code of a software is examined in various ways to detect potential misbehaviour, often without the need to explicitly execute it.

In this report we provide guidelines for the selection and use of static analyses for the verification of the safety and security properties of software only. Testing approaches to achieve these goals are beyond the scope of this document. However, this does not mean that static and dynamic analyses are unrelated activities. In fact, often static analyses can unfold their true potential when combined with corresponding testing activities. Some aspects of how testing and static analyses depend on each other can be found in Deliverable D3.3 [2] of this project.

1.1 Structure of this document

In Chapter 2 we give an overview of static analyses and structure them into four levels. These levels reflect both our and other experts experience in applying these methods. Based on these four levels the methodology in Chapter 3 is derived as a simple methodology that consists of three main levels.

1.2 Background

The VESSEDIA project is bringing safety and security to many new software applications and devices. In the fast-evolving world, we live in, the Internet has brought many benefits to individuals, organisations and industries. With the capabilities offered now (such as IPv6) to connect billions of devices and therefore humans together, the Internet brings new threats to the software developers and VESSEDIA will allow connected applications to be safe and secure. VESSEDIA is proposing to enhance and scale up modern software analysis tools, namely the mostly open-source Frama-C Analysis platform, to allow developers to benefit rapidly from them when developing connected applications. At the forefront of connected applications is the IoT, whose growth is exponential and whose security risks are real (for instance in hacked smart phones). VESSEDIA is taking this domain as a target for demonstrating the benefits of using our tools on connected applications. VESSEDIA is tackling this challenge by:

1. developing a methodology that allows to adopt and use source code analysis tools efficiently and produce similar benefits as already achieved for highly-critical applications (i.e. an exhaustive analysis and extraction of faults),
2. enhancing the Frama-C toolbox to enable efficient and fast implementation,
3. demonstrating the new toolbox capabilities on typical IoT (Internet of Things) applications including an IoT Operating System (Contiki),
4. developing a standardisation plan for generalising the use of the toolbox,
5. contributing to the Common Criteria certification process, and
6. defining a label “Verified in Europe” for validating software products with European technologies such as Frama-C.

DRAFT

Work package 1 of the VESSEDIA project is concerned with the development and assessment of safety and security verification methodologies.

Whilst verifying properties on source code is necessary to ensure safety and security of industrial applications, it is also necessary to express and verify properties on models and link them to the code in order to ensure a high level of confidence in the whole industrial system. WP1 aims therefore at designing languages able to express models of systems, scenarios and their associated system-level properties. We are designing a high-level language and methodology for the specification of safety and security properties as software requirements. Software analysis techniques have been used for the quality assurance of security/safety critical systems. In order to achieve their widespread adoption for IoT, we are identifying the following issues:

- Provide guidance on security threats for different classes of IoT.
- Achieve a better understanding and exploitation of the benefits and drawbacks of various static analyses.

The cost structure of the developed methodology will be scalable in order to make it suitable for application fields that are very cost sensitive.

Choosing which static analyses to apply to a given software usually depends on business constraints (e.g., product risk classes and deadlines) as well as on the application domain. The aim of task 1.4 was to provide, with respect to benefits and efforts, a methodology to find a cost-efficient trade-off between basic and sophisticated analyses.

1.3 Existing guidance for static analyses

In order to make the need for a methodology to choose appropriate analyses more apparent we have a look at the standard EN 50128 for the development of safe software of railway systems. Static Analyses and Formal Methods in EN 50128 shows the relationships between those parts of the standard that are related to static analyses and in particular to formal methods. Items labelled with “A” refer to tables of “Criteria for the Selection of Techniques and Measures”. Items labelled with “D” refer to short descriptions of techniques.

DRAFT

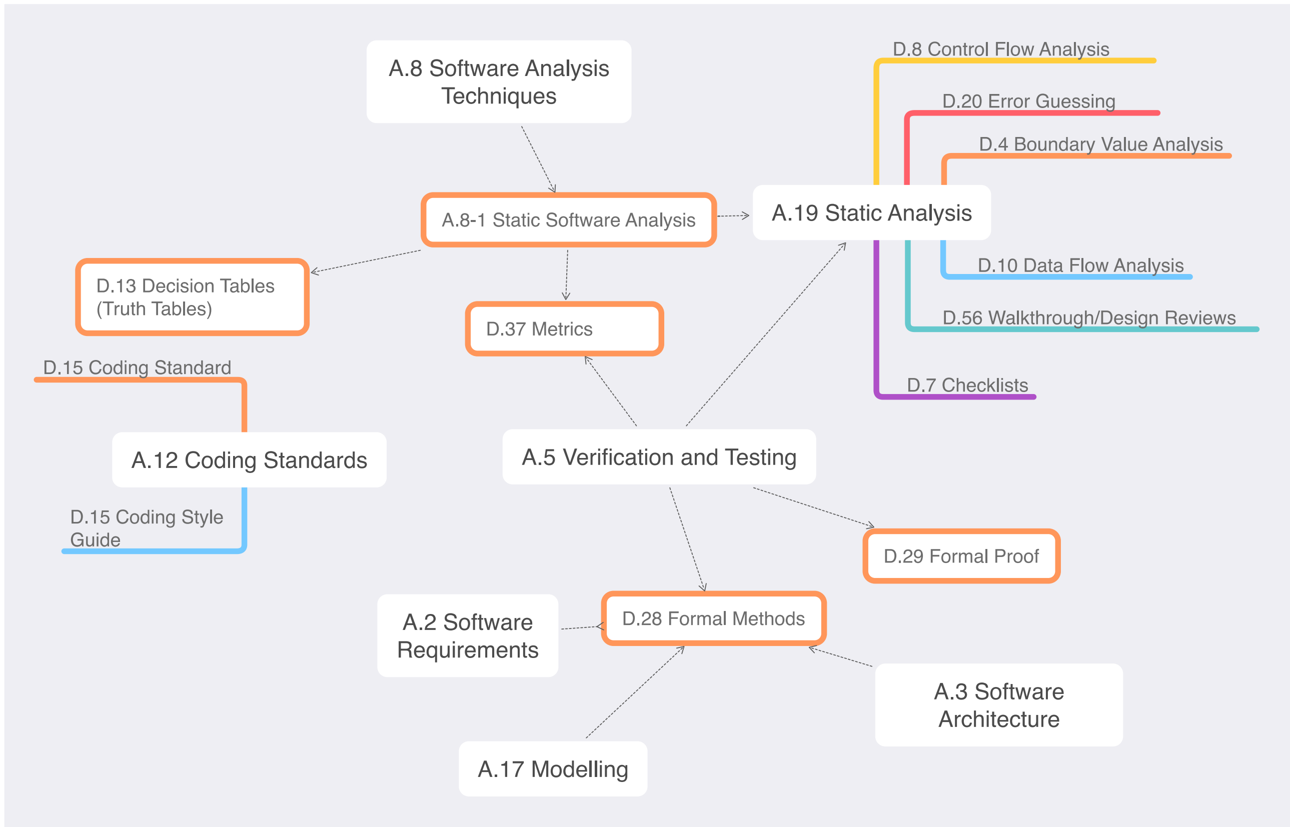
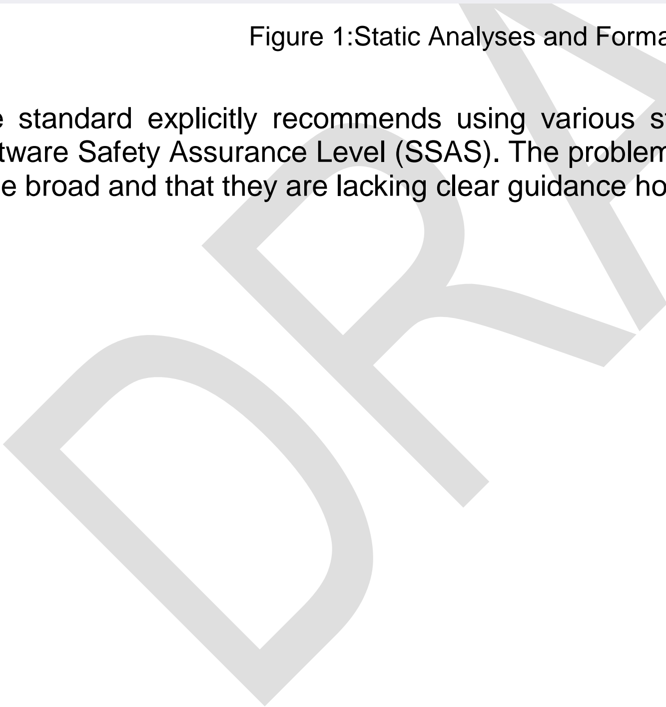


Figure 1:Static Analyses and Formal Methods in EN 50128

The standard explicitly recommends using various static analyses, depending on the intended Software Safety Assurance Level (SSAS). The problem, however, is that the recommendations are quite broad and that they are lacking clear guidance how they should be applied.



Chapter 2 A four-level classification of static analyses

We present in this chapter a classification of static analyses into the following four levels:
1. *basic* static analyses, such as compiler messages and warnings (strictest level)
2. *simple* static analyses, to find non-portable and suspicious program parts
3. *advanced* static analyses, explicitly enforcing given programming guidelines (beyond simple syntax/semantic restrictions)
4. *formal* static analyses, e.g., provably establishing the absence of run-time errors

In the following, we discuss each level in detail, giving typical application examples, assessing its usage of resources, both human and computational, and commenting on its main fields of impact.

Note that other authors suggest somewhat different hierarchies, often putting more emphasis on formal analyses.

Black et al. [NIST-8151, Sect. 2.1.1–2.1.4, p. 7–9] distinguish
1. Sound Static Program Analysis
2. Model Checkers, SAT Solvers and Other "Light Weight" Decision Algorithms
3. Assertions, Pre- and Postconditions, Invariants, Aspects and Contracts
4. Correct-by-Construction and Model-Based Development

All their levels are a refinement of our level 4.

For ADA SPARK[1] code, Moy [Kosmatov et al.] uses:
1. Stone level — Ada code adheres to the stronger SPARK rules
2. Bronze level — initialization and data flows are correct
3. Silver level — no run-time error can occur
4. Gold level — key integrity properties are proven

His level 1 roughly comprises our levels 1 and 2, while his levels 2, 3, and 4, are a refinement of our level 4.

2.1 Some general remarks on the combination of static analyses

Given a tool *X*, we define its *domain* to be the set of possible vulnerabilities that can be detected by *X*, and can't be detected by any other tool with less costs. Note that the domain of a tool also depends on the availability of other tools and their costs. From an abstract point of view, it is obvious that checking for each vulnerability using the cheapest tool will lead to minimum overall cost.

Similarly, from this abstract point of view, omission of any available tool with a non-empty domain will result in higher overall costs, and possibly even in a blind spot for some kinds of vulnerabilities. Particular care must be taken in order to have each vulnerability reported only by one tool, rather than by all that can detect it. For this reason, e.g. running all tools in parallel is disadvantageous. Instead, starting with the cheapest tool, each one has to be used individually, and all its findings have to be fixed before proceeding continuous with the next (more expensive) tool. This way, a more expensive tool won't report vulnerabilities found already by a cheaper one, since they are no longer present. Also, the amount of diagnostic tool output to be tracked down by the developer is reduced. When the quality assurance process has to be redone after changes to the software, ideally none of the vulnerability detection tools will report any new finding. In this case, the human effort is limited to initiating the run once for each tool. This approach depends, however, on a clear understanding of the capabilities and limitations of the used tools.

[1] ADA SPARK is an analysis approach for Ada programs which is comparable to Frama-C

DRAFT

2.2 Basic static analyses

The cheapest analysis level consists in full exploitation of all features of common everyday static analysis tools. Whatever information the employed compiler provides about the source program should be used; hence the strictest checking level should be chosen when running the compiler. This includes enabling all available kinds of warnings and hint messages. In order to be of any use, these messages should be followed, that is, the source code should be improved such that they disappear in subsequent compilation runs. The number of unavoidable messages should be reduced as far as possible, in order to minimize the human effort of reading through all messages after every compiler run. Ideally, there should be no messages at all, such that a newly appearing one will attract attention immediately.

While these measures should go without saying, our experience implies that often they do not, even in safety-critical software development projects. More specifically, we have often encountered the situation where a customer tasked us with conducting a static analysis with Frama-C. Once we started the analysis, we observed that the customer hadn't been bothered at all by the fair amount of compiler warnings!

2.2.1 On strong type checking

Strong type checking is called “one of the most pervasive applications” by a recent NIST3 report on software vulnerabilities to the White House Office of Science and Technology Policy [BBGF16, Sect. 2.1.7, p.10]. Extensive application of user-defined types can help to find “semantic errors” like e.g. a wrong parametrization order of an API call and wrong typing of an array or pointer access.
Unfortunately, both the C and C++ programming language do not support this kind of typing. Although “typedef” allows for user-defined types, they are handled in a transparent way. Neither C or C++ will issue any warning on potential misuses of type aliases.
We therefore recommend to follow a *strong typing approach* in designing critical software. For the application domain of embedded systems, it is of particular importance to deal with physical units, such as *meters*, *seconds*, etc., and the usual combinations thereof, to express e.g. velocity or acceleration figures. Concepts for strong type checking in such a setting have been devised long ago [3], based on systems of *dependent types*. They allow to detect that “v*t” has the *length* type if “v” and “t” has the *velocity* and *time* type, respectively, while “v+t” is an erroneous expression in that case. This approach can also help to prevent compatibility errors when using both SI and non-SI units. A (in)famous example for such an error is the loss of the ***Mars Climate Orbiter***[2] in 1999. Nowadays there are publicly available libraries that allow to incorporate the precise SI units into the application code, for example the C++ ***Boost.Units*** library[3].

2.3 Simple static analyses

The lint utility attempts to detect features of C program files that are likely to be vulnerabilities, to be non-portable or to be wasteful. It also performs stricter type checking than does the C compiler. The Lint tool flags some suspicious and non-portable constructs, likely to be vulnerabilities, in C language source code. Meanwhile the term “lint” has been

[2] See https://en.wikipedia.org/wiki/Mars_Climate_Orbiter
[3] See https://www.boost.org/doc/libs/1_69_0/doc/html/boost_units.html

DRAFT

generalized to arbitrary programming languages, meaning heuristic tools checking for suspicious language usage.

According to its documentation, the FreeBSD lint[4] checks for the following possible problems:

- unreachable statements
- loops not entered at the top
- unused variables
- logical expressions with constant values
- assignments to variables of smaller types
- casts of "questionable portability"
- "unusual operations" on enums
- gcc extensions
- vulnerabilities (according to some heuristic tests)
- non-ANSI C
- non-traditional C
- undefined, but used functions and external variables
- undefined structs

Moreover, it checks function calls for inconsistencies, such as

- unused function arguments
- mixed returns with and without value
- functions called with varying numbers of arguments
- parameter type mismatch in function call
- unused function return values
- using the return value of a void function

Modern compilers can often detect many of the constructs traditionally warned about by lint. Writers of lint-like tools have continued to improve the range of suspicious constructs that they detect. We show here a simple example of where a "lint-like" program can detect suspicious code that appears perfectly normal to a compiler.

```
int foo(int a)
{
    int b = 0;
    if (a > 4) {
       b = 0;
    } else {
        b = 1;
    }
    b = 5;
    return b;
}
```

[4] See release 11.0 on https://www.freebsd.org/cgi/man.cgi

DRAFT

When compiling this program with 'cc -Wall -c foo.c' no warning is reported. This is perfectly right; the problem though is that the highlighted part of the function does not affect the function's return value '5' at all. The analysis tool FlexeLint[5], however, correctly points out the following:

Previously assigned value to variable 'b' has not been used

Such a message can be an important contribution to ensure a high degree of code quality.

2.4 Advanced static analyses

As mentioned in the beginning of this chapter, we consider methods as *advanced static analyses*, when they are enforcing given programming guidelines that go beyond the most basic syntax/semantic restrictions. Over the years, various *domain-specific* programming guidelines have been published to document best programming practices to ensure the safety and/or security of software.
Often it is not clear how certain syntactic checks can make the software more robust. For example, some guidelines require that each occurrence of "{" or "}" in the source code appears on a separate line. On the other hand, it is known that explicitly requiring braces around certain code fragments as shown in the following code fragment

```
// instead of
if (isFoo)
    isFoo = false;

// use braces
if (isFoo) {
    isFoo = false;
}
```

can help avoiding nasty errors. In particular, the well-known "goto fail" vulnerability[6] in Apple's SSL implementation could have been prevented with this type of static analysis.[7]

Among the various programming guidelines that are used in industry we mention here

- High Integrity C++ Coding Standard [4]
- Joint Strike Fighter Air Vehicle C++ Standards [5]
- MISRA C and C++ Coding Standards [6]
  AUTOSAR C++ 14, Guidelines for the use of the C++14 language in critical and safety-related systems [7]
- SEI CERT C Coding Standard [8]
- SEI CERT C++ Coding Standard [9]

From a practical point of view, it can be said that the line between simple and advanced static analyses becomes more and more blurred. This is the reason why in our methodology in Chapter 3 we will merge these two categories in a new one that we refer to as heuristic static analyses. This terminology is in accordance with the one from the NIST report [9].

[5] See http://www.gimpel-online.com/OnlineTesting.html
[6] See https://nvd.nist.gov/vuln/detail/CVE-2014-1266
[7] See https://dwheeler.com/essays/apple-goto-fail.html

2.5 Formal static analyses

The highest level of static analyses, with respect to both analytical power and cost, are formal methods and tools. Their strength rests on rigorous mathematical reasoning. Nowadays more and more tools are available that rely in one form or another on formal methods. One of several challenges when applying formal methods is the need for a *formal specification* of the properties that are supposed to be formally verified. Thus, formal methods are difficult to apply when the desired properties are only vaguely understood. On the other hand, sometimes the necessary formal problem description can be derived without too much involvement from the user. One example of such a formal method is *abstract interpretation* for proving the absence of undefined behaviour. We will discuss in some detail in Section 2.5.1.

2.5.1 Abstract interpretation

The abstract interpretation method performs symbolic execution of source code to accumulate information about the set of possible values a variable may take [CC76, CC77]. Such a set is again represented in a symbolic way, e.g. as an interval [l...h] of integers. Based on this information, it is possible to warn about program constructs that may get input values beyond their domain of definition, such as *division by zero* or *out of bound array indices*. In C or C++ programs, both examples result in *undefined behaviour* and are a well-known source of security vulnerabilities. As the conditions for these potentially dangerous program constructs are defined in the C/C++ standard there is no need for the user to explicitly specify them. At the same time, the more precise the program's input data can be specified the more precise and faster the source code can be *abstractly interpreted*.

When a set can't be represented exactly, an appropriate superset is chosen; hence, an *over approximation* is computed. As a consequence, this method is *sound*, i.e. when it doesn't report a possibly undefined operation, it is guaranteed that there is none. This behaviour distinguishes abstract interpretation from all testing approaches, as well as from static analyses that merely rely on *heuristics*.
According to Doyle, [[10] p.3], abstract interpretation is "superior to current software development practices in terms of coverage, scalability and benefit for the effort. Within the VESSEDIA project, the Frama-C *EVA* plug-in offers abstract interpretation functionality. As a drawback, due to its reliance on over approximations abstract interpretation will usually report *false positives*, i.e. warn about operations being possibly undefined which in fact are not. To avoid this kind of imprecision in simple cases, Frama-C represents small sets explicitly, rather than as an interval.

Imprecisions may occur also due to variable dependencies when an own value range is computed for each variable. As a consequence, a *false positive* about a possible read of an uninitialized variable will be issued. To reduce the number of this kind of imprecisions, Frama-C and other tools offers several forms of *relational representations* of variable dependencies.

2.5.2 Hoare-style verification

The *postcondition verification* method uses mathematical theorem proving to establish user-given formal properties of a program [Hoa69].

This method differs from the *abstract interpretation* approach in needing extensive information from the user; in exchange, it provides considerable expressive power with respect to the properties that can be specified. As mentioned before, while abstract interpretation is done largely automatic, it can handle only a limited set of pre-defined properties. As an example, given a program to sort database records, abstract interpretation can demonstrate that no run-time exception can occur, but it needs

verification to establish that its output is always ordered ascendingly and neither longer nor shorter than its input; the latter properties would have to be specified, in a formal language, by hand.

Within the Frama-C verification platform, the WP plug-in supports this verification approach. Besides a specification (“contract”) for each function, a typically large number of formal annotations are necessary in the source code to guide the verification process. For example, for every loop the program loop invariants must be formulated. In addition, the employed theorem proving tools are usually unable to verify sophisticated code properties, requiring them to be broken down manually into simpler pieces; to this end, intermediate formulas (“assertions”) need to be placed in the code. Note that none of these annotations influence the actual run-time behaviour of the software, contrary to the C library *assert* mechanism they appear as (special) comments i.e. as invisible to a compiler (but still recognizable by Frama-C).

Due to the high costs related with this formal verification approach, we don’t explain it here in more detail. Our reason for mentioning it at all is that we will suggest a more cost-effective specialization in Section 2.5.3.

2.5.3 Hoare-style verification restricted to run-time errors

It is of course good practice to address all diagnostic messages from analysis tools and, where possible, to modify the *software under analysis* such that no warnings are issued by future tool runs.

Unfortunately, this is not always possible with respect to abstract interpretation tools. As a limited means to help Frama-C avoiding false positives in future analysis runs, hints may be given by some kinds of assertions. For example, asserting a disjunction may cause Frama-C to make an according case distinction [[11]Sect.7.1.2, p.76]. However, not all kinds of false positives can be eliminated this way.

As an alternative, we investigated the use of Hoare-style verification proofs (Section 2.5.2), restricted to run-time errors. We refer to these method as *minimal contracts* because the corresponding Hoare-style contracts would not address the full functionality of a function. The goal of this approach, we wish to emphasize, is not to replace abstract interpretation tools but rather to complement them. Also note that a big advantage of abstract interpretation tools is their good scalability with respect to program size. In general, abstract interpretation is applied to complete programs, whereas Hoare-style verification (including minimal contracts) works more on the level of individual software components.

The Frama-C plug-in *RTE* (run-time errors) can be used to automatically generate the necessary assertions, thus saving the user a lot of manual effort.

Additional loop invariants are, of course, needed whenever a variable’s value is influenced in a loop and used therein or thereafter. However, we expect these invariants to be relatively simple, in many cases just stating lower and upper bounds of variables. Ideally, Frama-C could suggest simple loop invariants based on a run of its value analysis plug-in which computes this information, anyway. By automatically inserting initial versions of such annotations into the source code, which the user later could fine-tune in case this is needed by the WP plugin. We consider this kind of additional annotations as helpful when trying to understand the code; therefore, we suggest to keep them permanently as some additional documentation. If this doesn’t meet a user’s point of view, Frama-C could easily provide functionality to remove them.

DRAFT

Chapter 3 Suggestion of a three-step methodology

In this chapter we present our three-step methodology for applying a wide range of static analyses to assure the security and safety of software. While the VESSEDIA project is mainly concerned with software for IoT systems, this methodology is not necessarily restricted to it.

A particular challenge of the IoT domain, however, is that there is a strong emphasis on quick development cycles and nifty system features and less so on quality. To be quite clear, whatever static analysis is applied, it cannot be a remedy for badly understood requirements and misconceptions on software security.

The three steps of our methodology address, nevertheless, cost concerns by starting with the proper use of diagnostic messages of compilers (Section 3.1) and continues with tools for checking established programming guidelines (Section 3.2). The final step consists in applying *sound static analyses* (Section 3.3), a term that better reflects the essential aspect of what is also known as *formal methods*.

An important idea is that these three steps can and should be applied one after another. Doing it this way ensures that their increasing capabilities and cost of application are applied in the most meaningful way.

3.1 Proper use of compiler diagnostics

The first step of our methodology relies, as trivial as it may sound, on the proper use of the diagnostic messages of compilers. This step is also the easiest and cheapest step since developers are quite familiar with the capabilities of compilers. Still, it is useful to mention some important points.

Regarding cost, we mention that in many cases compilers for popular programming languages are available free of charge. However, using these free tools properly (for example following the points mentioned below) is related with certain costs. These costs are mostly related with understanding the proper usage of the tools and fixing the reported problems. On the other hand, compiling a file with strict diagnostics does not really increase the build time.

3.1.1 Chose proper language version

In abuse of language, one often speaks of *the* C programming language or *the* C++ programming language. However, both languages, and other programming languages as well, have evolved and major evolution steps can be identified by respective *language standards*. There are often also language *dialects* that are specific to certain application domains or compiler vendors. Specific international standards are sometimes identified by the year in which a standard has been published. As there are sometimes not only extension but also subtle changes in the semantics we highly recommend to explicitly specify the language version. For the gcc and clang compilers, for example, one can use the option `-std=c++14` in order to refer the C++ ISO standard published in 2014.

3.1.2 Enable warning options

We recommend using a reasonable strict set of compiler warnings when compiling the source code. The **gcc**[8] and **clang**[9] compilers, for example, provides the option *Wall* to enable many useful warnings. This is in many cases a reasonable choice. Note, however, that the *Wall* option does not cover all warning options. Thus, it makes sense to explore the manual and look for warning options

[8] GCC, the GNU Compiler Collection, https://gcc.gnu.org
[9] Clang, A C language family frontend for LLVM, https://clang.llvm.org

that are useful for the product under development. Some other useful options outside the scope of *Wall* are covered by the options *Wextra* and *pedantic*.

3.1.3 Discover more warning options

The clang compiler provides an option *Weverything* that enables *all warnings*. This option can be used to *search* for useful warning options and enable those for later production runs. In other words, it is not recommended to use this option as default warning option.[10]

3.1.4 Enforce fixing reported warnings

Next, it must be ensured that the code is actually fixed when warnings are reported, so that finally, no warnings occur at all. One way to enforce this is to turn warnings into errors. The clang and gcc compilers provide the option *Werror* to achieve this goal.

DRAFT

[10] See https://quuxplusone.github.io/blog/2018/12/06/dont-use-weverything/ /

3.2 Deployment of heuristic static analyses

When we speak of *heuristic* static analyses, we mean it in the sense as it is used by the authors of NIST-8151 **[9].** We also use this term to merge the two categories of *simple static analyses* (Section 2.3) and *advanced static analyses* (Section 2.4).

Using these tools is for several reasons more expensive than just using the diagnostic capabilities of compilers. First, tools that rely on heuristic static analyses are often not freely available. Second, it can take more time to understand and properly configure the tools. Thirdly, conducting the analyses often takes more time than a simple compilation. Finally, the diagnostic messages of a heuristic analysis tool might be *inconclusive* because the specific heuristics flag issues that are not really problems. Heuristic analyses share this reporting of *false alarms* with sound analyses. However, the problem is more prevalent with sound analyses.

As we explained in Section 2.4, heuristic methods are often regularly applied to enforce certain programming guidelines. Our methodology therefore requires that one decides which programming guidelines should be followed.

3.2.1 Select and understand appropriate programming guidelines

The first and most important task when using heuristic static analyses is to select and understand the underlying programming guidelines or properties to be analysed. It does not make much sense to analyse whether the source code follows MISRA-C [6] rules if the developers do not know what these rules mean and imply. On the other hand, it might well happen that new set of rules become suddenly important, for example because *security* becomes suddenly an issue in domains that were mainly concerned with *safety*.

3.2.2 Identify heuristic static analyses tools that support the selected guidelines

Once the rules have been understood and taught it is important to select appropriate tools. This step must include sufficient training in using the tools in order to ensure that developers and quality assurance experts can use the tools properly. These costs can be considered to be higher than those related to the proper use of compiler diagnostics.

3.2.3 Enforce fixing reported problems

As in the case of Section 3.1.4, it might sound trivial to demand that problems discovered during a heuristic analysis are indeed addressed and resolved. We explicitly mention this step in our methodology in order to prevent that these more advanced static analyses are used merely as a *fig leaf.*

3.2.4 Reconcile with compiler diagnostics

At the same time, tool users should be aware that there can be an overlap between the capabilities of stricter compiler diagnostics and heuristic static analyses. If this is the case, we recommend to use the corresponding compiler options as early as possible. Here again it might be necessary to explore which warning options are provide by the compiler (see Section 3.1.3).

3.3 Application of select sound static analyses

The final and most challenging step of our methodology consists in applying *sound static analyses* to ensure the various safety or security properties of software. We use here the term *sound analyses* instead of *formal analyses* for several reasons. Formal methods, that is methods that are based on logic and mathematical theories, are around for decades. One of their most important features is that they can provide *assurance that comes from a chain of logical* reasoning [[9], Sect. 2.1.1, p.7]. This is the meaning of *sound.* The term *formal,* on the other hand, can have in everyday usage a fairly negative meaning in the sense of *ritualistic* or *inflexible.*

3.3.1 Understand scope of sound static analyses

Applying sound static analyses can require a considerably deeper knowledge of mathematical logic than simpler static analyses. Before these analyses are applied, it is important to understand and evaluate the potential of these methods.

3.3.2 Select appropriate analysis tools

Once, appropriate methods have been identified it is necessary to select suitable tools that rely on these methods. Note that in practice it can appear as if it is the other way around because often *tools* are sold, not *methods.*

3.3.3 Apply sound analyses after simpler analyses have been conducted

The application of sound static analyses requires usually considerably more time than heuristic static analysis. One reason for this are false positive alarms which often occur at a considerably higher rate than when using purely heuristic methods. For this reason, it is highly advisable that sound methods are only applied once the potential of compiler diagnostics and heuristic methods have been exhausted.

DRAFT

Chapter 4 Conclusions

This document presents a three-level methodology for applying static code analyses to ensure the safety and security of software on IoT devices. The three levels group together related techniques for static source code analysis of increasing capabilities. At the same time, the various levels also roughly reflect the different costs related with these methods. Our methodology, thus, also allows to weigh up to a limited degree the strength of a static analysis method against its cost of application. For a more detailed analysis of the *cost* of applying static analyses we refer to Deliverable 1.6 [1].

While this report only considers static analysis techniques it is important to keep in mind that a proper quality assurance strategy for IoT systems must combine static analyses *and* dynamic analyses (i.e. testing techniques). These issues are addressed in Deliverable 3.3 [2] of the VESSEDIA project.

DRAFT

Chapter 5 Bibliography

[1] VESSEDIA Deliverable D1.6, Economic rational and metrics report of the effectiveness and efficiency of the use of VESSEDIA outcomes
[2] VESSEDIA Deliverable D3.3, Guidelines for combination of static and dynamic analyses
[3] Jochen Burghardt. Concepts for an extended type checker for the Z specifi- cation language. Arbeitspapier 995, GMD, Jun 1996.
[4] Integrity C++ Coding Standard, https://www.perforce.com/resources/qac/high-integrity-cpp-coding-standard
[5] Joint Strike Fighter Air Vehicle C++ Standards, http://www.stroustrup.com/JSF-AV-rules.pdf
[6] MISRA C and C++ Coding Standards, https://www.misra.org.uk
[7] AUTOSAR C++ 14, Guidelines for the use of the C++14 language in critical and safety-related systems, https://www.autosar.org/fileadmin/user_upload/standards/adaptive/17-03/AUTOSAR_RS_CPP14Guidelines.pdf
[8] SEI CERT C Coding Standard, https://resources.sei.cmu.edu/downloads/secure-coding/assets/sei-cert-c-coding-standard-2016-v01.pdf
[9] SEI CERT C++ Coding Standard, https://resources.sei.cmu.edu/downloads/secure-coding/assets/sei-cert-cpp-coding-standard-2016-v01.pdf
[10] NIST-8151: Dramatically Reducing Software Vulnerabilities: Report to the White House Office of Science and Technology Policy
[11] Richard Doyle. Formal methods, including model-based verification and correct-by-construction. In Software and Supply Chain Assurance (SSCA) Working Group Summer, McLean/VA, Jul 2016
[12] David Bühler, Pascal Cuoq, and Boris Yakobowski. EVA — the evolved value analysis plug-in. User's Manual Silicon-20161101, CEA LIST, Software Reliability Laboratory, 2016.
[13] N. Kosmatov, C.Marché, Y. Moy, J. Signoles: Static versus Dynamic Verification in Why3, Frama-C and SPARK 2014

DRAFT