D2.5

Basic Analyzers and collaboration of analyses - Final Release

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 / D2.5 / 1.0
Work package contributing to the deliverable:	WP2
Due date:	December 2019 - M36
Actual submission date:	15th of January, 2020

Responsible organisation:	KUL
Editor:	Bart Jacobs
Dissemination level:	PU
Revision:	1.0

Abstract:	This report describes further enhancements to the front-ends and specification libraries developed in Task 2.1, the improved analyzers developed in Task 2.2, and the prototypes of new analyzers developed in Task 2.4 and 2.5
Keywords:	software verification, software analyzers

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

Editor
Bart Jacobs(KUL)

Contributors
Virgile Prevosto (2-CEA)
Gergely Eberhardt (4-SLAB)
Bart Jacobs (8-KUL)

Executive Summary

This document reports on further work done as part of all tasks in WP2 during M33-M36. The work was centered around the following four technologies:

- Extensions of Frama-C basic and derived analyzers (MdR and StaDy plug-ins)
- ACSL++ and Frama-Clang, which extend the ACSL specification language and the Frama-C verification framework for C programs with support for C++ programs
- The VeriFast tool for modular formal verification of C and Java programs
- The SecuRate metric for prioritizing verification alarms

It is a complement to deliverables D2.3 and D2.4, submitted at M32 and does not cover all the tools that were released in these two previous deliverables, but only those on which major further work was done during M33-M36.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Today's software industry faces the considerable challenge of delivering software for powering IoT devices and applications that meets both the market's stringent cost and time-to-market constraints, and users' expectations of privacy and security. Indeed, a number of well-publicized cases of IoT devices being hacked, allowing attackers to spy on users or even inflict physical damage, indicate that current industry practice fails to meet these goals.

The VESSEDIA project aims to address this challenge by developing methodologies, metrics, and tools that enable industry actors to more cost-effectively assess, verify, and certify the security of the IoT software they develop. Work Package 2, in particular, collects the VESSEDIA activities concerning verification tool development. Within Work Package 2, Tasks 2.1, **Programming Languages Artefacts**, 2.2, **Extensions of Existing Analyses**, and 2.3, **Runtime Vulnerability and Attack Protection through Data and Control Flow Integrity Enforcements** are concerned with the development of individual verification tool building blocks, whereas Tasks 2.4, **Composite Analysis Tools**, and 2.5, **Integration into the Software Development Lifecycle**, are concerned with composing building blocks to achieve advanced functionalities.

In the current state of the art of verification tools, no tool is capable of offering the strongest security guarantees without any user intervention. Therefore, all currently proposed tools make a trade-off between the strength of the security properties verified, and the amount of user intervention required. Choosing the right tool then depends on weighing the cost of security issues versus the cost of verification in the context of a particular software development project: for a project where the risk of residual security vulnerabilities remaining in the delivered product is relatively tolerable, tools that require little or no user intervention are preferrable, whereas for a project where such residual vulnerabilities would be catastrophic, tools that require significant user expertise and effort but provide high assurance of security are justified.

To cover this spectrum and in this way support all types of software development projects, VESSEDIA verification tool development has focused on the following four tools, supporting three modes of operation:

- *Frama-C Eva* is an *abstract interpretation tool*, built on the *Frama-C* framework for verification tools for the C programming language. It analyzes a program's source code for particular types of security properties, by computing an abstraction that overapproximates the program's set of reachable states. In contrast to dynamic analysis, if successful it can guarantee the absence of certain types of vulnerabilities in all possible executions of the program. The user intervention required is moderate: it is limited to tuning the abstraction mechanism. Its verification power is limited by the fact that

it supports a fixed set of built-in abstractions, which means that it supports only the security properties that can be expressed using those abstractions and the programs that can be analyzed using those abstractions.

- *Frama-C WP* is a *deductive verification tool*, also built on Frama-C. In contrast to abstract interpretation, it is *modular*: it verifies, for each of the program's modules separately, that the module satisfies the security properties specified by the user. Compared to abstract interpretation, it can provide stronger guarantees and support more complex programs, at the cost of requiring the user to supply a specification for each module, as well as other user-defined abstractions such as loop invariants and data structure specifications.

- *Frama-C StaDy* is based on the collaboration between static and dynamic analysis tools. More precisely, static analyzers can prove that a program is correct with respect to some properties of interest, but when a potential issue is raised, some further (manual) work is indeed to decide whether a real bug has been spotted or this is a false alarm, consequence of the abstractions that the analyzer has made in order account for all possible concrete executions of the programs. On the other hand, dynamic analyzers based on sets of test cases always report real issues, but the set of possible inputs is usually so large that it is not feasible to explore it exhaustively, meaning that some issues might go unnoticed by a test campaign. The main goal of StaDy is thus to focus the attention of test case generators on potential issues emitted by Eva or WP, trying to find an input vector that will indeed trigger the issue.

- *VeriFast* is another deductive verification tool. Whereas Frama-C WP is based on classical Hoare logic, VeriFast is based on an extension of Hoare logic called *separation logic*. Separation logic forces the user to specify *ownership* of all memory locations and other resources used by the program, further increasing the amount of user intervention required; in return, this allows VeriFast to support more advanced programming constructs such as, in particular, concurrency. VeriFast supports the C and Java programming languages.

The Frama-C tools[1] and VeriFast[2] are relatively mature tools that have existed for a long time; the development done as part of VESSEDIA has consisted in making significant improvements to these tools, enhancing their applicability to IoT verification use cases in important ways.

The structure of the remainder of this report is as follows:

- Chapter 2 reports on the work done in the latest Frama-C version on StaDy, and Markdown-Report, a plug-in to generate analysis reports in formats suitable for consumption by external tools or for integration in more general evaluation reports.

- While many IoT applications are written in C, at least as many are written in C++. Chapter 3 reports on the work done to extend the Frama-C framework to also support the C++ programming language. This involves in particular developing ACSL++, an extension of Frama-C's ACSL notation for C program annotations, as well as Frama-Clang, a Frama-C module that translates ACSL++-annotated C++ programs to ACSL-annotated C programs. The latter is based on the parser and typechecker of the popular Clang C++ compiler.

[1] `https://frama-c.com/`
[2] `https://github.com/verifast/verifast`

- Chapter 4 reports on the work done in the context of the **VeriFast** tool for modular formal verification of C and Java programs. Specifically, it reports on initial results in the verification of liveness properties of the I/O behavior of concurrent programs.
- All of the VESSEDIA verification tools generate so-called *alarms* when potential security issues are detected. An important challenge for users is to decide which alarms to investigate and address first. Chapter 5 reports on the work implementing **SecuRate**, a metric for prioritizing verification alarms based on the number of affected products and the number of affected devices.

Chapter 2

Frama-C

Frama-C 20.0 Calcium was released on December 4th, 2019 and forms the basis of this deliverable with respect to the extensions to the StaDy and Markdown-Report plug-ins, as well as to the Frama-C kernel itself, that have been made since D2.3 and D2.4. The main changes with respect to these deliverables are detailed in this chapter.

2.1 Frama-C Kernel

The improvements to handling ghost code that were available through the Medium plug-in presented in Chapter 4 of D2.3 have been incorporated directly in the main type-checker of Frama-C and are now in a stable state.

In addition, the incorporation of the Markdown-Report plug-in into the main repository of Frama-C (see below), led to a consolidation of the internal representation of Markdown documents. While mostly visible only to developers at this stage, this will result in a easier production of such documents in the whole platform (notably for the Report plug-in itself), leading to a smoother integration of Frama-C results into various reports.

2.2 Markdown-Report

The Markdown-Report plug-in, which was made available in an experimental version as part of D2.4, has been stabilized and incorporated into Frama-C main distribution, while its Markdown internal representation has replaced the one previously available in Frama-C itself (see above).

2.3 StaDy

In order to facilitate the use of StaDy over CEA use-case in WP6, various improvements have been made over the plug-in. The most important ones include:

- preliminary support for recursive types
- preliminary support for bitwise operations in ACSL formulas
- preliminary support for pointer subtraction in ACSL formulas

- simpler instrumentation of the code, and better support for C standard library specifications as found in Frama-C standard headers

2.4 Installation

The StaDy archive that is part of this deliverable is meant to be compiled against the Frama-C 20.0 Calcium version (which is also included). As was the case in D2.4, it depends on the PathCrawler plug-in, though, which is not publicly distributed.

The Frama-C archive included in this deliverable is exactly the same as the one that is available for download on the `https://frama-c.com` website. Hence, it is possible to install it through the opam package manager, which is the preferred installation method as indicated in the `INSTALL.md` file in the archive. This file also contains detailed installation instructions, as well as the necessary dependencies, allowing for a manual installation.

Once Frama-C itself has been installed, StaDy itself can be installed, provided PathCrawler is available, by decompressing the archive, and, in the `Frama-C-StaDy` directory, issuing the commands:

```
./configure
make
make install
```

Chapter 3

ACSL++ and Frama-Clang

Deliverable D2.3 provided the first version of the ACSL++ manual, describing a formal specification language for C++, based on ACSL for C. A new version, taking into account discussions and feedback from Vessedia partners is part of D2.5. It does not provide brand new features, but include many clarifications over the constructions that were proposed in the initial version.

In parallel, some improvements were made to the frama-clang plug-in that provides Frama-C with a C++ front-end. These improvements are publicly available as frama-clang 0.0.8, and include:

- compatibility with Frama-C 20.0 Calcium and Clang 9.0;
- support for the `\exit_status` construction in ACSL++;
- better support for ghost code at frama-clang level, in conjunction with the enhancements mentioned in the previous chapter;
- various fixes allowing the C++ part of the plug-in to be compiled against `-Wall` option of the `g++` compiler.

Chapter 4

VeriFast

During M33-M36, KUL has started to investigate the problem of formal modular verification of liveness properties of the I/O behavior of concurrent programs. For example, for a concurrent web server, we want to verify that it responds to each request, or, in other words, that it does not *starve* any clients. Often, web servers are multithreaded, to exploit hardware parallelism. Such programs typically terminate only under fair scheduling assumptions. This complicates liveness reasoning.

While our research on this topic is ongoing, we have obtained initial results. The document *Verifying Termination of Busy-Waiting for Program Abortion*, attached to this deliverable, describes an approach for verifying termination under fair scheduling of programs where some threads abort the program and other threads run forever. Since verifying that a program eventually performs some I/O action X can be encoded as verifying that the program terminates, assuming that action X aborts the program, this is a step towards our goal.

We are aiming to submit a paper on our work to the prestigious ECOOP 2020 conference.

Chapter 5

SecuRate

See the separate document attached to this deliverable.