D4.1

Metrics for VESSEDIA tools in quality assurance

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 / D4.1/1.0
Work package contributing to the deliverable:	WP 4
Due date:	Jun 2018– M18
Actual submission date:	2nd July, 2018

Responsible organisation:	SLAB
Editor:	Gergely Eberhardt
Dissemination level:	PU
Revision:	1.0

Abstract:	In this deliverable we propose and define metrics, which can be helpful during the verification and evaluation process. For each metrics we specify implementation guidelines for Frama-C developers, but the metrics ideas can be used in other tools such as VeriFast also.
Keywords:	Internet of Things, Metrics, Verification tools, Common Criteria

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

Editor

Gergely Eberhardt (SLAB)

Contributors (ordered according to beneficiary numbers)

Virgile, PREVOSTO (CEA)

Dillon, PARIENTE (DA)

Regina, BIRO (SLAB)

Cédric, BERTHION (AMO)

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.

Executive Summary

This deliverable describes the metrics identified for VESSEDIA tools in quality assurance. Metrics are crucial for assessing safety and security assurance and enable developers to prioritize critical bugs to be fixed - moreover, they also contribute to the assessment of security evaluation and/or certification effort. Some metrics presented in this document are newly defined for the Frama-C plugins while others are improvements for the already existing metrics, such as code coverage. Further metrics will provide qualitative data on the progress of code analysis and indications of the severity and/or criticality of the identified vulnerabilities in a scored way. The document first gives an overview about metrics in general and in the context of Common Criteria. After the definition, the proposed metrics are described along with related plugins and implementation guides. Finally, the metrics which are already present in Frama-C are described, focusing on the integration with the proposed metrics.

Contents

Chapter 1 Introduction 1
1.1 VESSEDIA motivation and background 1
1.2 Structure of the document..1
1.3 Related deliverables...1
Chapter 2 Metrics in the security and safety assurance process (AMO)..................2
2.1 Metrics overview 2
2.2 Metrics in Common Criteria...2
Chapter 3 VESSEDIA metrics 4
3.1 SecuRate 4
3.1.1 Integration with Frama-C 5
3.1.2 Design of the SecuRate service...5
3.1.3 Fingerprint techniques overview 6
3.1.4 Implementation details..9
3.2 CriticalDepth...9
3.3 Liveness Metrics 12
3.4 Size Definition Distance Metrics..13
3.5 Dangling Pointers Persistence Metrics..14
3.6 Cryptographic Secrets Persistence Metrics 14
3.7 Static Analysis Coverage Metrics...15
3.8 Quantitative assessment for deductive verification tools..............................16
3.9 “CWE scoring of an alarm” Metric 17
3.10 “Criticality of an alarm” taint analysis Metrics...18
3.11 “Statistics” Metric 18
Chapter 4 Metrics in Frama-C 21
Chapter 5 Summary and Conclusion 22
Chapter 6 List of Abbreviations..23
Chapter 7 Bibliography 24

List of Figures

Figure 1: SecuRate integration with Frama-C 5
Figure 2: SecuRate service architecture 6
Figure 3: Call graph of the Apache apr_memcache resource 10
Figure 4: Alarms in the function `ms_find_conn` 11
Figure 5: Call graph on the faulty function `ms_find_conn` 11
Figure 6: Example of a program in pseudo code 12
Figure 7: Control Flow Graph of the program defined in Figure 6 12
Figure 8: Example of a buffer overflow in C 13
Figure 9: Example of a buffer overflow in C 13
Figure 10: Example of a User-After-Free in C 14
Figure 11: Example of a C program using cryptographic secret 15
Figure 12: Code coverage example 15
Figure 13: Code and specification sample 16
Figure 14: Always-true contract 16
Figure 15: Metrics for pre-conditions 17
Figure 16: Post-conditions metrics 17
Figure 17: Taint analysis example 18
Figure 18: Metrics over original code 19
Figure 19: Metrics over normalized code 20

Chapter 1 Introduction

1.1 VESSEDIA motivation and background

The VESSEDIA project aims to bring 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 proposes 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 will take this domain as a target for demonstrating the benefits of using our tools on connected applications. VESSEDIA will tackle this challenge by 1) developing a methodology that allows to adopt and use source code analysis tools efficiently and produce similar benefits than 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.

1.2 Structure of the document

After these introductory sections, Chapter 2 gives an overview about metrics in general and in the context of Common Criteria. Chapter 3 defines the proposed metrics with examples and some implementation guidelines. After the proposed metrics, Chapter 4 describes the current metrics in Frama-C. Finally, Chapter 5 concludes the document and paves the way for the subsequent work in the project.

1.3 Related deliverables

Based on the metrics definitions described by this deliverable, implementation activities will be performed in WP3. The implemented metrics will be used during further work in the use-case evaluation in D4.6 [3] and quality tests in D4.5 [2].

Chapter 2 Metrics in the security and safety assurance process (AMO)

2.1 Metrics overview

Source code analysis is used by two different kinds of users during the life-cycle of a product: developers and security evaluators. Source code analysis is used for testing and bug tracking by developers. A good coding behaviour is to integrate some source code analysis tools to the project. Once an analysis tool is set up, it can be used all along the project on each version of the code. Developers have full knowledge of their product so they can finely tune the analyser to avoid false positives. Remaining false positives can be checked and documented in order to make the next analysis always more precise. Thus, source code analysis is also part of source code maintenance in the life-cycle of a product.

In another hand, security evaluator uses source code analysis to find vulnerabilities in a product. They are using source code analysis tools like "vulnerability scanners". They do not have a full knowledge of the product. Depending of the documentation they have been provided by the developers, reverse engineering has to be performed to understand the role of each module of the project. Thus, evaluators cannot check the whole code of a program because it would be too time consuming. They have to focus on critical modules where critical vulnerabilities are more likely to be. In the same way, analysers' outputs have to be as synthetic as possible. The goal of a security evaluator is not to discover every vulnerability in the product. Detecting one critical vulnerability may be enough to set a verdict for the evaluation and to consider the product as non-secure.

Note that a security evaluator has to prove that a vulnerability can be exploited and that it is indeed a security issue for the product. In the main case, developers do not need to know if a potential bug can be exploited or not. Safety processes require that developers should patch the code and fix every known bug.

To sum up, developers need source code analysers to avoid as many bugs as possible, whereas security evaluators need automatic tools to quickly find some vulnerabilities even if they are missing some of them. This is one of the main differences of process between verification and evaluation.

Depending on the user's objectives metrics in static analysers will not be used in the same way. General metrics about the code like the number of lines of code (LOC), the cyclomatic complexity, numbers of warning per lines of code etc. are relevant as indicator of the code quality and can be very useful for project management. To fix bugs and detect vulnerabilities, metrics which extend warning information are the most valuable. These metrics enable developers to choose which warnings are important or which warnings can be quickly fixed. The same kind of metrics is used by security evaluator to focus only on bugs that can lead to security flaws.

2.2 Metrics in Common Criteria

In Common Criteria, no specific metrics on source code are defined from the evaluator point-of-view. No list of metrics is provided and no Common Criteria Component directly depends on source code metrics that evaluators have to check.

However, source code metrics are referred in part 3 of Common Criteria[1] in class ALC which is about “Life-cycle support”. Some examples are given as source code complexity metrics, defect density (errors per size of code) or mean time to failure. These metrics are not about security, they are more about safety and code maintenance. They do not directly refer to attacks or to vulnerabilities but they can help evaluators to understand how developers are working, how they are testing their product. Evaluators are more likely to trust developers that can provide metrics which reveal that the target has been heavily tested. Thus, providing this kind of metrics increases the assurance in the security of the product even if they do not directly address security issues.

For example, the assurance in the security of the product is increased when developers can provide metrics that prove that they have a high coverage for their testing campaign.

Metrics are also used in Common Criteria during the Independent Testing (ATE_IND) Independent Vulnerability Analysis (AVA_VAN) performed by the evaluator. During this part of the evaluation, the source code of the product can be audited. Evaluators are using code analyser to speed up their work and to automate it as much as possible. The main difficulty is to categorise the warning returned by the tools. Which warning is reliant and which one is a false positive? The evaluator has to manually check the warnings to be sure and this is very time-consuming. Thus, the more metrics an evaluator has to qualify a warning, the more he is able to select valuable issues and the more efficient his work is.

[1] Common Criteria Documentation is available at https://www.commoncriteriaportal.org/cc/

Chapter 3 VESSEDIA metrics

During development, verification and evaluation process a large number of alarms can be raised by the different components of the Frama-C. Some alarms are important, some alerts have less relevance and some other alerts are false positives. If the number of these alarms is too high, the developer or the evaluator will not be able to check every single alarm one by one. Thus, there should be some quantitative data, which make possible to sort the alarms by their severity or criticality and finally the developers and evaluators can focus on the most severe and critical problems first.

Every calculation of severity and criticality values is based on subjective categories. There is a constant effort to make these assessments as objective as possible. One of the most acknowledged calculation method is the Common Vulnerability Scoring System[2] (CVSS), which is currently at version 3.0. Even the CVSS tries to assess the severity of a vulnerability in a qualitative and objective way: in case of the meltdown and spectre[3] vulnerabilities the CVSSv2 resulted scores from 2.0 to 5.1, while the CVSSv3 resulted scores from 6.7 to even 8.1[4], which means very high range. Therefore in VESSEDIA, we do not aim to develop an automated way of vulnerability and alarm scoring in an objective way, but we try to collect metrics, which correlates with the severity or criticality of the vulnerability in some way and giving the opportunity to the developer and the evaluator to use the most relevant metrics supporting the actual task.

3.1 SecuRate

SecuRate metrics is aimed to measure the criticality of an alarm in terms of the estimated number of affected products and the estimated number of affected devices. In general a complex software solution uses a lot of libraries, reused codes and open source software projects. In case of the IoT-related code, the code reuse is also an increasing trend, since it is required to implement complex functionality within a very limited time and resources.

We identified the following tasks, which should be performed by the SecuRate plugin:

- **Identifying known vulnerabilities** using vulnerability databases such as CVE and NVD and using internal database from previous problems and alerts.
- **Estimate the severity of the alarm** by assessing the number of affected products.

For example, the following code from the `dnsmasq` (version below 2.78)[5] is vulnerable to a stack-based buffer overflow as we analysed in detail in D2.1.

```
/* RFC-6939 */
if ((opt = opt6_find(opts, end, OPTION6_CLIENT_MAC, 3)))
{
    state->mac_type = opt6_uint(opt, 0, 2);
    state->mac_len = opt6_len(opt) - 2;
    memcpy(&state->mac[0], opt6_ptr(opt, 2), state->mac_len);
}
```

The above code was compiled into the `dnsmasq` if the DHCPv6 was enabled and used, but in most of the cases the IPv6 is supported in modern IoT devices. Although the vulnerability was disclosed at September 27th 2017, the latest version of `dnsmasq` in our firmware database was

[2] https://www.first.org/cvss/

[3] https://meltdownattack.com/

[4] https://www.scademy.com/2-10-cvss-not-the-meltdown-youd-expect/

[5] https://security.googleblog.com/2017/10/behind-masq-yet-more-dns-and-dhcp.html

2.76 and the latest firmware of various routers contained the vulnerable `dnsmasq` with `DHCPv6` enabled.

3.1.1 Integration with Frama-C

The metrics generated by the SecuRate requires a large database, which contains binary and source fingerprints and the collected alarm and vulnerability information pieces. Although the databases and the related functionality can be deployed to the same infrastructure than the source code analysis, in most cases it is better to collect the required information in a centralized way. Because of this, the service functionality is separated from the integration with the source code analysis as it is presented in the Figure below.

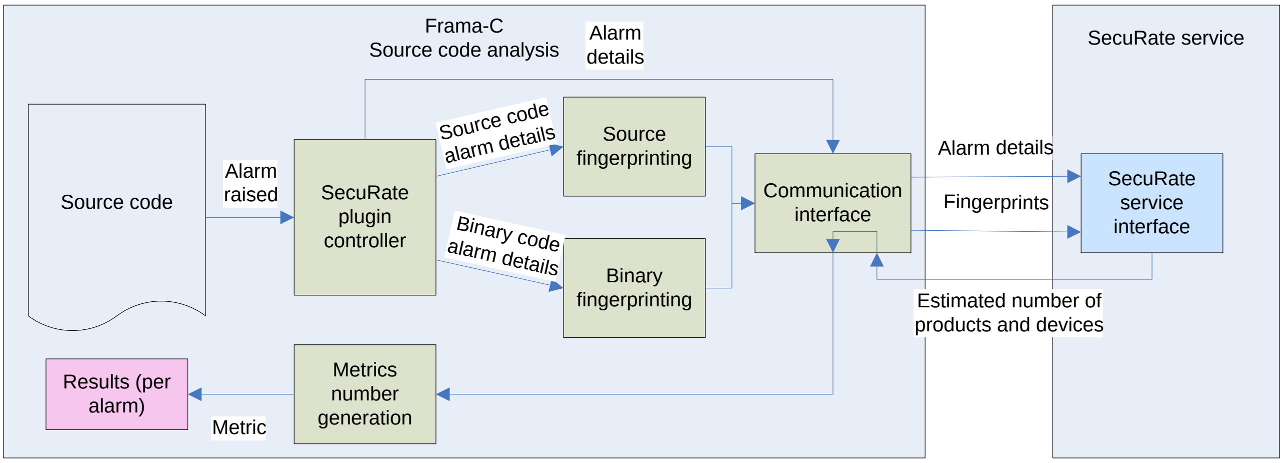


Figure 1: SecuRate integration with Frama-C

The SecuRate plugin, which make the connection possible between the SecuRate service and the Frama-C, has the following components:

- *SecuRate plugin controller*: The plugin controller is responsible for collecting every relevant information details about alarms raised by other analyser plugins, such as EVA or WP.
- *Source and binary fingerprinting*: Depending on the availability of the source and binary code, the main features of the function, component or library are extracted and sent to the SecuRate service via the Communication interface.
- *Communication interface*: The SecuRate Frama-C plugin and the SecuRate service should communicate with each other via a secure channel provided by the Communication interface.
- *Metrics number generation*: Based on the received data from the SecuRate service the Metrics number generation component will calculate a metrics in proportion to the relevance of the module or code part.

3.1.2 Design of the SecuRate service

The SecuRate service is designed to perform the following tasks:

- Identifying the binary or source code based on the fingerprints. For this task an internal firmware and source code database should be created and maintained.
- Collect new vulnerabilities or alarms to the internal vulnerability database in order to use this information to check existing vulnerabilities and to measure the quality of other modules using the same component.
- Identifying every products/devices which contains the same component, which was identified based on the fingerprint.
- Estimate the number of remotely accessible devices, which uses the identified component. For this estimation SecuRate service will use open-source intelligence (OSINT) techniques, such as IoT search engines, next to the internal databases.
- Collect and process new firmwares and source codes from various sources.

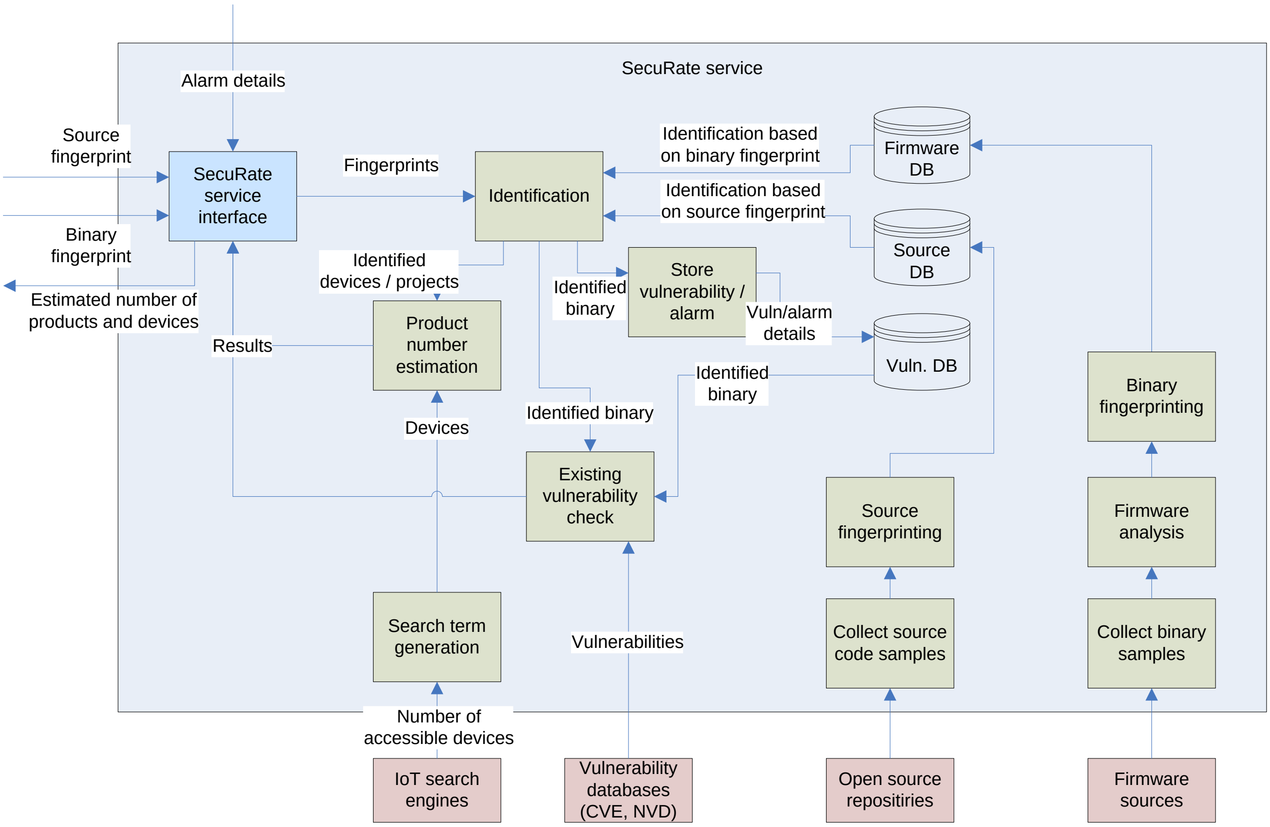


Figure 2: SecuRate service architecture

3.1.3 Fingerprint techniques overview

The SecuRate metrics uses fingerprints in order to find existing vulnerabilities by comparing its input to vulnerable sources. Fingerprinting algorithms, in general, map arbitrary large data items to bit strings of a given length, which uniquely identify the original data. This section describes the existing fingerprint techniques and provides rationale on which approaches could be integrated with SecuRate.

Since the SecuRate metrics aims at identifying common vulnerabilities in binary and source codes as inputs and then estimates the number of affected devices and products, not all fingerprinting methods can be considered. For example, plain hash functions cannot solve this problem, as even a one-bit change to a given file would result in a completely different hash value. The remaining options consist of advanced hash functions, like rolling hash and fuzzy hash; comparison tools which discover and analyse differences between the functions of two given binary or source codes; and other solutions, such as comparing strings existing in two input files. The sections below describe such existing solutions and implementations which could be integrated to SecuRate.

Hashing schemes

The hashing schemes which have the property that a small change to the file being hashed results in a small change to the hash are called similarity digests. There are several different approaches, such as feature extraction, Locality Sensitive Hashing (LSH) schemes and Context Triggered Piecewise Hashing (CTPH) schemes.

- ***ssdeep***[6]
  The `ssdeep` hash is a similarity digest which is a de facto standard in the area of malware analysis, and is an implementation of the CTPH scheme, also called as fuzzy hashes. The

[6] https://ssdeep-project.github.io/ssdeep/index.html

scheme operates by segmenting the file and then evaluating a 6 bit hash value for each segment. Then, as a similarity measure, ssdeep calculates the edit distance between the digests.

- ***sdhash***[7]
  `Sdhash` created a similarity digest by first identifying features with low empirical probability, then hashed these features into a bloom filter and encodes the bloom filter as the output digest. The similarity score is calculated by a normalized entropy measure between the two digests.
- ***Nilsimsa***[8]
  The `Nilsimsa` hash implements a bit sampling LSH scheme, using the hamming distance between the digests as a similarity measure.
- ***TLSH***[9]
  `TLSH` (Trend Micro Locality Sensitive Hash) is a fuzzy matching library. For comparison, it computes 35 bytes long hash values where the first 3 bytes capture the information about the file as a whole (such as length, etc.), while the last 32 bytes capture information about the incremental parts of the file.
- ***Rolling hash***
  Usually used for comparison of strings and pattern matching, a rolling hash is a hash function where the input is hashed in a window that moves through the input – thus the name - , allowing new hash values to be rapidly calculated from the previous hash values. There are several implementations of rolling hashes in different languages, such as `rollinghashcpp`[10] and `rollinghashjava`[11].

Data comparison schemes

Data comparison schemes calculate and display the differences and similarities between data, typically used to compare source or binary code. Named after the Unix diff utility, the implementations, methods and results of data comparison are called diffs. Below a list of tools are described which could be useful for the development of SecuRate.

- ***BinDiff***[12]
  `BinDiff` is a comparison tool for binary files - as an add-on to IDA-, widely used by security professionals to find similarities and differences in disassembled code, for example to analyse vulnerability patches, malware variants, etc.
- ***Turbodiff***[13]
  `Turbodiff` is a binary diffing tool, developed as a plugin for IDA. It discovers and analyses differences between the functions of two binaries.
- ***Rizzo***[14]
  Rizzo is an IDA plugin that identifies and renames functions between two or more IDA database files (IDBs) based on formal signatures, references to unique strings and unique constants, fuzzy signatures and call graphs.
- ***Lscan***[15]
  `Lscan` is a tool which identifies libraries in statically linked/stripped binaries. It can be used for recognizing common functions in compiled binaries and for determining what libraries

[7] https://github.com/sdhash/sdhash
[8] https://github.com/rholder/nilsimsa
[9] https://github.com/trendmicro/tlsh
[10] https://github.com/lemire/rollinghashcpp
[11] https://github.com/lemire/rollinghashjava
[12] https://www.zynamics.com/bindiff.html
[13] https://www.coresecurity.com/corelabs-research/open-source-tools/turbodiff
[14] https://github.com/devttys0/ida/blob/master/plugins/rizzo/rizzo.py
[15] https://github.com/maroueneboubakri/lscan

they are using. It uses FLIRT (Fast Library and Recognition Technology) signatures to perform the library identification.

- ***Kam1n0***[16]
  `Kam1n0` tries to solve the efficient subgraph search problem (i.e. graph isomorphism problem) for assembly functions. Given a target function it can identity the cloned subgraphs among other functions in the repository. Kam1n0 supports rich comment format and has an IDA Pro plug-in to use its indexing and searching capabilities via IDA Pro.
- ***FCatalog***[17]
  `Fcatalog` finds similarities between different binary blobs. It has two parts: server and client – the client side of `Fcatalog` is an IDA plugin that can compare newly acquired binary functions to previously reversed binary functions.
- ***Diaphora***[18]
  `Diaphora` is a program diffing plugin for IDA Pro and Radare2, very similar to `BinDiff` and `Turbodiff` (see above).
- ***BinSourcerer***[19]
  `BinSourcerer` is an Assembly to Source Code matching framework for IDA Pro, written in Python. `BinSourcerer` was intended to recreate the functionalities that RE-Google was providing before the Google Code Search API was discontinued. The plugin can be used for code search (e.g. on GitHub), function tagging and generates a disassembly feature file which can be used in binary analysis.
- ***Cardinal***[20]
  `Cardinal` (CPC Aggregation by reversing and Dumping in Arrays Lightweight) is a tool that can find similarities even between binaries compiled with different optimization flags or completely different compilers, in order to overcome malware compiling variations.
- ***Nucleus***[21]
  `Nucleus` is a tool integrable with IDA Pro, based on the paper "*Compiler-Agnostic Function Detection in Binaries*"[22], published at EuroS&P 2017.
- ***radiff2***[23]
  `Radiff2` is a feature of `radare2` offering offset-based function diffing. By default it shows the bytes that were changed and the corresponding offsets, but it is also capable of computing the distance and the percentage of similarity between two files.
- ***Multidiff***[24]
  `Multidiff` aims to provide differences between a large set of objects by doing diffs between relevant objects and displaying them in a sensible manner, handy when looking for patterns.

Other

This section describes solutions that fallout from the first two schemes, but they are relevant for fingerprinting and similarity matching.

- ***Qira***[25]

[16] https://github.com/McGill-DMaS/Kam1n0-Plugin-IDA-Pro
[17] https://www.xorpd.net/pages/fcatalog.html
[18] https://github.com/joxeankoret/diaphora
[19] https://github.com/BinSigma/BinSourcerer
[20] https://github.com/syreal17/Cardinal
[21] https://bitbucket.org/vusec/nucleus.git
[22] https://syssec.mistakenot.net/papers/eurosp-2017.pdf
[23] http://radare.today/posts/binary-diffing/
[24] https://github.com/juhakivekas/multidiff
[25] https://github.com/geohot/qira

`Qira` (QEMU Interactive Runtime Analyser) is a timeless debugger, competitor to `strace` and `gdb`.

- ***Bap***[26]
  `Bap` (Binary Analysis Platform) is a reverse engineering platform that targets binaries. It is written in `Ocaml`, but has bindings to C, Python and Rust.
- ***ByteWeight***[27]
  `ByteWeight` is a tool that recognizes functions in binaries. It outputs function information by providing function start (lowest address of each function), function boundary (a pair of the lowest and the highest addresses) and function instructions (a list of addresses, where each address is a start of a disassembled instruction).
- ***Distorm***[28]
  `Distorm` is a binary stream disassembler library project. It provides details of the disassembled instructions – hence called decomposer - which can be used for advanced binary code analysis.
- ***Capstone***[29]
  `Capstone` is a disassembly framework which is also a decomposer – it provides details and semantics of the disassembled instruction, such as a list of implicit registers read and written.

3.1.4 Implementation details

The SecuRate service will be capable of handling different hashing and comparison schemes implemented in separate plugins. Thus the user of the SecuRate service will be able to choose the most appropriate schemes for the actual task. As a baseline framework, SecuRate will use the FACT[30] framework developed by Fraunhofer FKIE, which provides a multitasking environment with the capability of initial firmware unpacking and analysis.

3.2 CriticalDepth

A metrics named `CriticalDepth` is prototyped by DA in the context of WP4. Its goal is to present, in the GUI of Frama-C, the call stack depth for each alarm generated by EVA, given the fact that an alarm may be generated by several different potential execution paths.

This metrics permits to assess the reachability of the given alarms in terms of callstack depth. The depth is computed from the entry point of the analysis (by Frama-C plugin EVA).

It provides the verification team with an evaluation of the reachability from data and statements supposed to be potentially under the control of an attacker (namely from the entry point of the library or the whole application). Thus, the criticality of identified vulnerabilities is evaluated in terms of depth. A large depth would mean more effort to build an exploit scenario for a given weakness in the source code, from an attacker point of view (of course, this is heuristics, and does not apply to all cases!). On the contrary, the developers will be more inclined to focus - as a first intent - on shorter execution paths, then focusing more rapidly on the origin of the weakness, in practice potentially common to all of the other longer and vulnerable call paths.

This metric is implemented inside a Frama-C plugin (EVA) and its results are displayed through the EVA’s GUI module. Further improvements could be made in the context of WP3/T3.4 if needed, in

[26] https://github.com/BinaryAnalysisPlatform/bap
[27] http://security.ece.cmu.edu/byteweight/
[28] https://github.com/gdabah/distorm
[29] https://github.com/aquynh/capstone
[30] https://fkie-cad.github.io/FACT_core/

order to merge and synthetize the measurements done by the plugin to present them as conveniently as possible to the user.

As an illustration, the figure below presents the whole call graph of the Apache `apr_memcache` resource (analyzed from the `apr_memcache_incr` function by Frama-C):

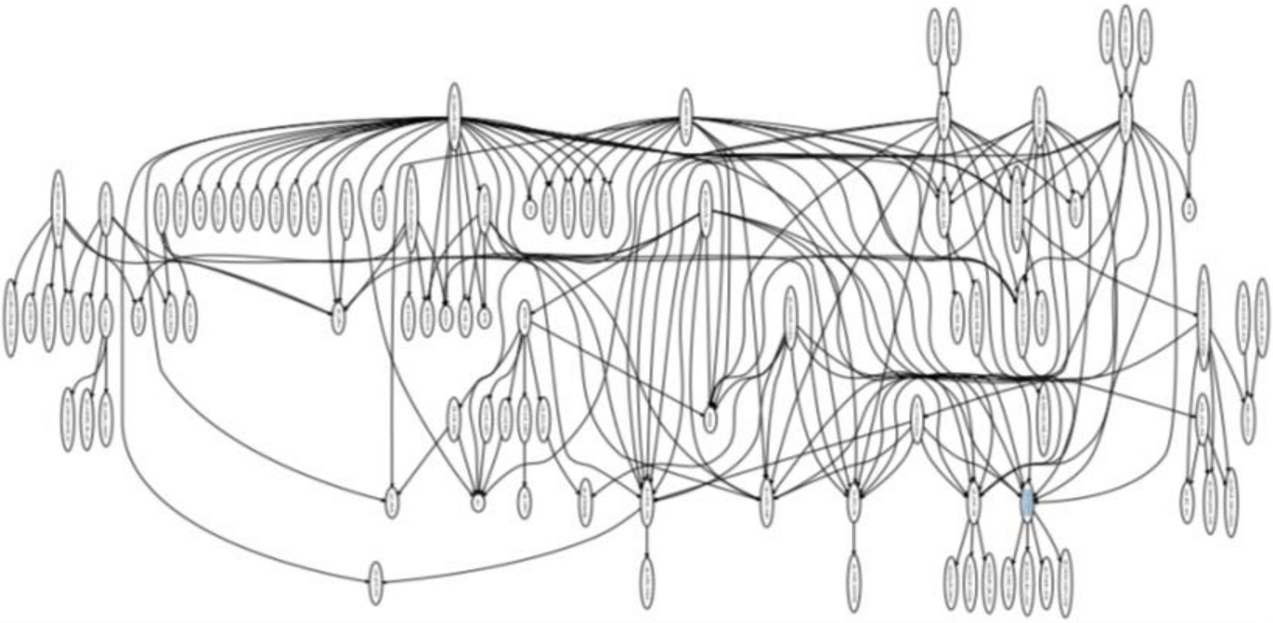
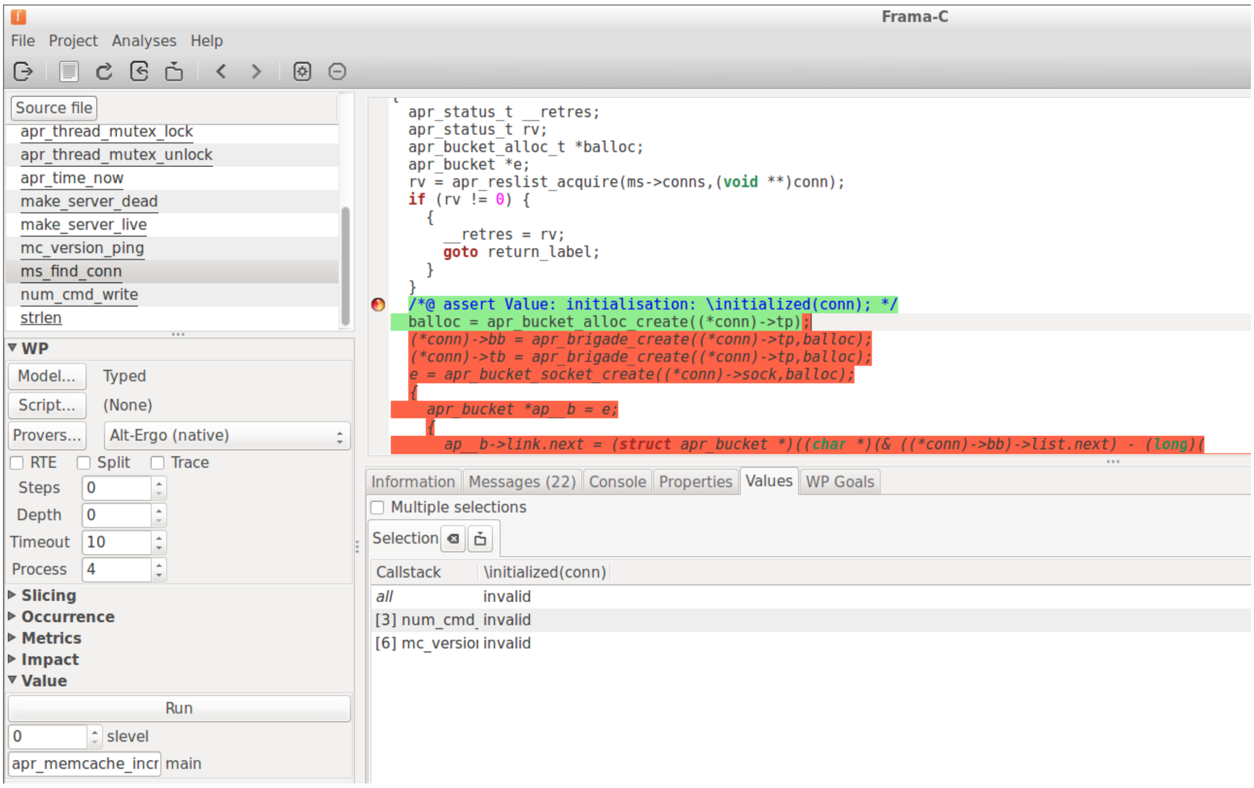


Figure 3: Call graph of the Apache apr_memcache resource

When analyzing this module, several alarms are raised. Some of them can be considered as critical, their status “INVALID” meaning that the execution will systematically lead to runtime errors. The figure below presents the function `ms_find_conn`, in which a statement handling the conn variable raises an alarm about its lack of initialization. In the `Callstack` column in the `Values` panel, the depth of the corresponding call stack is indicated by the CriticalDepth metrics plugin.

The first call stack has a depth of 3 (displayed between brackets), and the second one is 6. The user is obviously invited to first explore the 3-long call stack issue, in order to potentially find more rapidly the origin of the flaw (which could be also the same origin for the second, longer, path in the call stack).



```
{
  apr_status_t __retres;
  apr_status_t rv;
  apr_bucket_alloc_t *balloc;
  apr_bucket *e;
  rv = apr_reslist_acquire(ms->conns,(void **)conn);
  if (rv != 0) {
    {
      __retres = rv;
      goto return_label;
    }
  }
  /*@ assert Value: initialisation: \initialized(conn); */
  balloc = apr_bucket_alloc_create((*conn)->tp);
  (*conn)->bb = apr_brigade_create((*conn)->tp,balloc);
  (*conn)->tb = apr_brigade_create((*conn)->tp,balloc);
  e = apr_bucket_socket_create((*conn)->sock,balloc);
  {
    apr_bucket *ap__b = e;
    {
      ap__b->link.next = (struct apr_bucket *)((char *)(& ((*conn)->bb)->list.next) - (long)(
```

Model... Typed
Script... (None)
Provers... Alt-Ergo (native)
☐ RTE ☐ Split ☐ Trace
Steps 0
Depth 0
Timeout 10
Process 4

0 slevel
apr_memcache_incr main

☐ Multiple selections

Callstack	\initialized(conn)
all	invalid
[3] num_cmd_	invalid
[6] mc_versioı	invalid

Figure 4: Alarms in the function ms_find_conn

The figure below zooms in the call graph on the faulty function ms_find_conn (with an Invalid status alarm). Several execution paths are possible, from apr_memcache_incr and ms_find_conn functions, before reaching the dangerous statement:

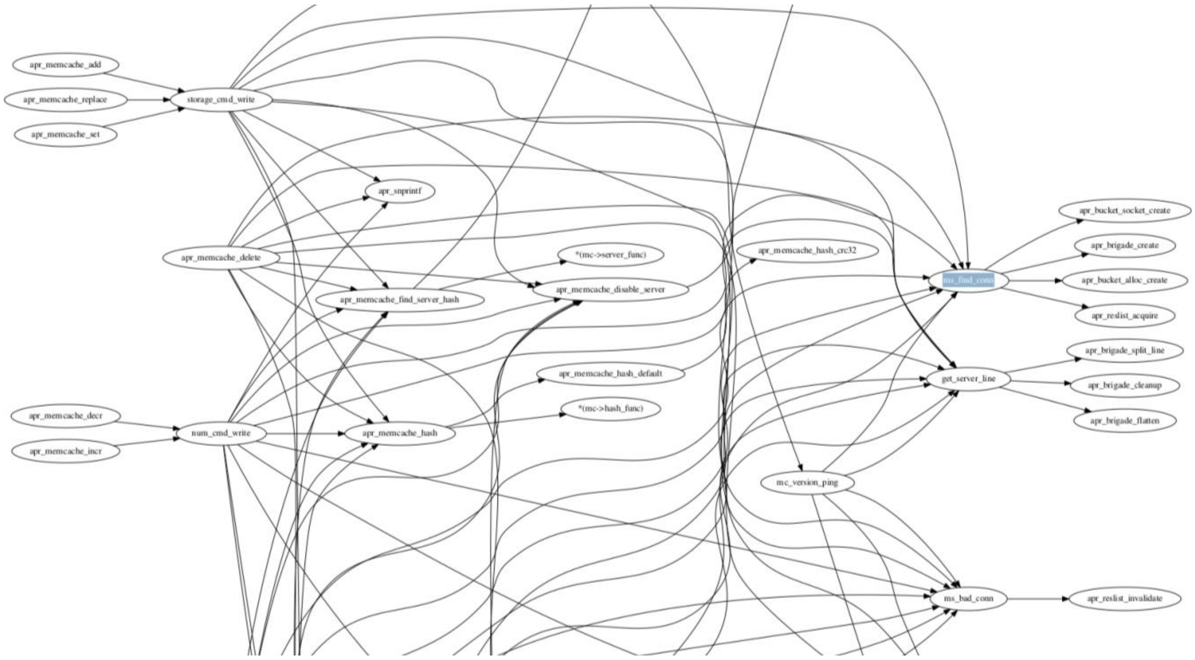


Figure 5: Call graph on the faulty function ms_find_conn

The metrics will then just help developers deal first with the shortest path from the entry point of the function, to the function/statement on which the alarm was raised.

3.3 Liveness Metrics

From a security point of view, a bug can be considered as a vulnerability if it is related to user inputs. When a warning is raised by Frama-C, the analyst has to check the origin of the data involved. Thus, one metrics to characterize a warning is raised by Frama-C is, for each data, a list of instructions which handled this data from its definition to its use in the suspicious instruction. Let’s consider the following typical program:

```
z = get(user_input) // z is set from a user input
if(z)
     if (z < 10)
        y = <value> // <value> is a constant integer value
    else
        y = z+1
    ptr = y
    *ptr // the pointer ptr is used for a writing, a reading or an
         // execution
else
    …  // some junk code (block E and F in Figure 2)
…. // some junk code (block G in Figure 2)
```

Figure 6: Example of a program in pseudo code

In our example, `*ptr` means that a pointer `ptr` is used in a line of code for a reading, a writing or an execution. An alert can be raised if this pointer `ptr` is dereferenced and if it targets an invalid memory region.

This program gets some data from the user and set them in the variable `z`. Depending on the value of this input, the variable y will be initialised with the value `z+1` or with a constant integer. Then, the pointer `ptr` is initialized with the value of `y`.

It has the following graph:

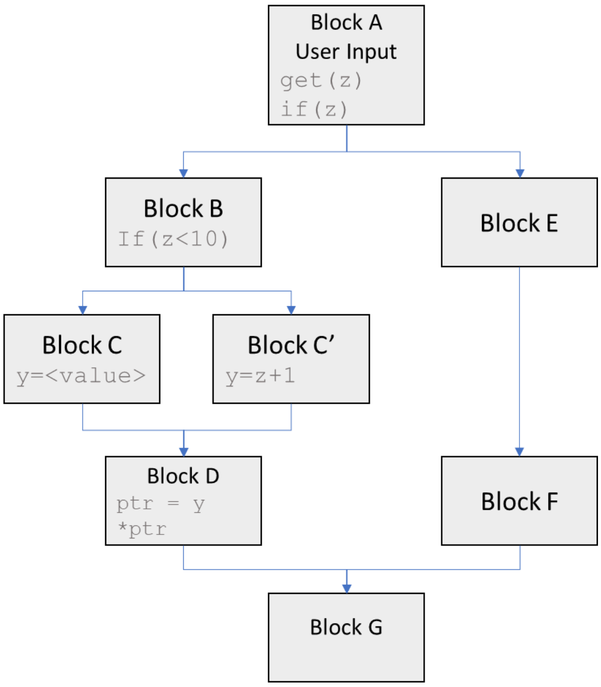


Figure 7: Control Flow Graph of the program defined in Figure 6

Each block is a set of lines of code which are always iteratively executed.

Let’s consider that an alert is risen by Frama-C in block D for the instruction `*ptr`. Thanks to the liveness metrics, an analyst would be able to know which instructions have modified the value of the pointer `ptr`. Thus, there will be no need to check in blocks E and F.

A simpler metrics can be the number of instructions that have handled the data involved in an alert since the data definition.

There are two ways to interpret this metrics. The first is to consider that the lesser instructions there are between the use and the definition of some data, the more likely an alarm using this data is a vulnerability.

Indeed, the relation between the original inputs and the data is likely to be all the more complex that the data are handled by several lines of code. If the relation is too complex, Frama-C may fail to reconstruct it and there may be a false positive.

Moreover, the simpler a vulnerability can be understood by an attacker, the easier it is for him to build an exploit. Therefore, simplest vulnerability should be promptly corrected.

As explain for the metrics CriticalDepth, if the distance between the definition of data and a crash is quite small, the work for the analyst to check the validity of a Frama-C warning is reduced. This is why warning with a low liveness metrics should be treated more urgently.

Technically, this metric can be implemented as a taint analysis based on the EVA plugin.

3.4 Size Definition Distance Metrics

The Size Definition Distance Metrics is the number of instructions between the definition of the size of a buffer and the use of this buffer.

For instance, let us consider the following code:

```
1.  int buffer[10];
2.  int iter = 0;
3.  int value = 1;
4.  for (iter = 0; iter < 12 ; iter++)
5.  {
6.      buffer[iter] = value; //warning overflow
7.  }
```

Figure 8: Example of a buffer overflow in C

In this example, there is a buffer overflow in line 6. Since buffer is defined in line 1, the Size Definition Distance Metrics would be 5, as there are 5 lines of code between the overflow warning and the definition of the vulnerable buffer.

For instance, let us consider the following code:

```
1.      void function0()
2.      {
3.          int buffer[10];
4.          … // 1000 lines of junk code
1004.       function1(buffer);
1005.       … // junk code
1998.   }
1999.
2000.   void function1(int * buffer)
2001.   {
2002.       int iter = 0;
2003.       int value = 1;
2004.       for (iter = 0; iter < 12 ; iter++)
2005.       {
2006.           buffer[iter] = value; //warning overflow
2007.       }
2008.   }
```

Figure 9: Example of a buffer overflow in C

In this example, there is a buffer overflow in line 2006. Since buffer is defined in line 3, the Size Definition Distance Metric would be 1005, as there are 1005 lines of code from the definition of the buffer to its misuse.

Thus, this metrics is quite similar to the Liveness Metrics. The smaller the distance is, the easier a buffer overflow warning can be checked. Since the size definition is quite close to the instruction that raises the warning, the number of lines of code that should be checked by an analyst is quite small too.

Moreover, if the distance between the definition of the size of a buffer and an instruction that raises a buffer overflow warning is large, there is a risk that the developer "forgot" the value of the size and made a mistake by using it.

Thus, if the value of this metrics is small, the warning should be quite simple to be checked by a human analyst and if the value is very large, it is more likely that a developer made a mistake when handling the size of the buffer.

Technically, this metrics can be implemented as a taint analysis based on the EVA plugin.

3.5 Dangling Pointers Persistence Metrics

Dangling pointers may be the cause of Use-After-Free and Double Free vulnerabilities. Any freed pointer should be erased as quickly as possible.

The Dangling Pointers Persistence Metrics is the number of instructions between the freeing of a pointer and the erasure of its value. A large value for this metrics means that this dangling pointer may be used by a large amount of instructions before being erased. Thus, this pointer can be considered as suspicious and it should be checked.

For instance, let us consider the following code:

```
1.    int * my_ptr = malloc(4*sizeof(int));
2.    … // code using my_ptr
3.    free(my_ptr)
4.    … // junk code without reallocating my_ptr
5.    my_ptr[2]=5; //User-After-Free
6.    … // junk code
7.    my_ptr = 0; //erasing the pointer
```

Figure 10: Example of a User-After-Free in C

In the code sample above, some data are set in the memory chunk pointed by `my_ptr` in line 5, but `my_ptr` was already freed in line 3. Thus, there is a User-After-Free vulnerability.

For multiple reasons, like code complexity, this User-After-Free may not be detected by static analysis. However, if the analysist can know in line 3, where the free is done, that the value of the pointer will be erased in the next lines of code, then the risk that the freed pointer will be reused is low. On the other hand, if the value of the Dangling Pointers Persistence Metrics is high, the pointer is still alive for a long period after its freeing, the risk that pointer is reused after being freed is higher.

Technically, this metrics can be implemented as a taint analysis based on the EVA plugin.

3.6 Cryptographic Secrets Persistence Metrics

Cryptographic secrets have to be erased from memory as soon as possible. Thus, their persistence in memory must be checked during a code audit.

The Cryptographic Secrets Persistence Metrics is the number of instructions between the last use of a cryptographic secret and its erasure from the memory. It is quite similar to the Dangling Pointers Persistence Metrics.

The larger this metrics is, the longer a value persists in memory and the more likely this value can be accessed from memory by an attacker. Thus, this value should be as small as possible.

For instance, let us consider the following code:

```
1.   int key = 0xdeadbeef;
2.   crypt_file(my_file, key);
3.   … // n lines of junk code
4.   key = 0;
```

Figure 11: Example of a C program using cryptographic secret

In the code sample above, the cryptographic secret is the variable `key` defined in line 1 and use in the function `crypt_file` in line 2. The variable `key` is no longer used, thus there is no reason that it is not erased just after the call to `crypt_file`. If the user annotated the variable key as a cryptographic secret in the analyser, the user would be able thanks the Cryptographic Secrets Persistence Metrics, to know the number of lines of code that separate the erasure of the key from its last use in the `crypt_file` function. If this value is too high, it is a sign that the cryptographic secret is not erased as soon as possible as it should be.

Technically, this metrics can be implemented as a taint analysis based on the EVA plugin.

3.7 Static Analysis Coverage Metrics

Abstract-interpretation based static analyzers like the EVA plug-in in Frama-C perform an abstract execution of the code starting from a given entry point and an abstract initial context. Depending on this entry point and the initial context, not all statements of the code under analysis may be visited during the execution. For instance, if we seek to analyze the function `f` shown in Figure 12, using either `main1` or `main2`, we will end up with different results. `Frama_C_interval` is a built-in that returns a non-deterministic result in the interval defined by its two arguments. Hence `main1` will only activate the first branch of `f`, while an analysis starting at `main2` will cover both branches of the function.

```
int f(int x) {
  if (x <= 4) return 0;
  else return 1;
}

int main1 () {
  int test = Frama_C_interval(0,4);
  return f(x);
}

int main2 () {
  int test = Frama_C_interval(0,10);
}
```

Figure 12: Code coverage example

Of course, vulnerabilities lying in branches that are not visited by an analysis will not be detected. It is thus important to be able to measure which proportion of the code under analysis actually has been covered by the analyzer. This metrics can be compared to the various coverage criteria that have been proposed for assessing the quality of test suites [1]. As it is the case with software testing, the most obvious an easily computable metrics, i.e the ratio between the number of statements covered and the total number of statements in the code under analysis might not always be the most relevant coverage measure. Indeed, if for instance the two branches of a given `if` statement have very different lengths, an analysis covering only the longest branch could achieve a pretty good coverage but still miss a critical vulnerability in the shortest branch. It is thus important to provide a set of coverage measurement that go beyond statement coverage. Branch coverage (ratio of the number of branches that have been considered by the analysis over the total number of branches in the code) is an interesting metrics in this respect. More generally, there exist many coverage criteria in the testing world, and a certain number of them would probably be relevant in abstract interpretation as well.

3.8 Quantitative assessment for deductive verification tools

Deductive verification tools attempt to prove that a given piece of code is conforming to some formal specification, usually expressed under the form of a function contract. As shown in Figure 13, such contract expresses what a function requires from its callers, and what properties it ensures when it returns back from having been called in a state which respects its requirements. In some circumstances, notably in the case of loops, additional annotations are needed and are to be proved as well as the main specification.

```
/*@ requires 0 <= x < 100;
    ensures \result == x * (x+1) / 2;
*/
int sum(int x) {
int res = 0;
/*@ loop invariant 1<=i<=x+1;
    loop invariant res == i * (i-1)/2;
    loop assigns i, res;
*/
for (int i = 1; i <=x; i++) { res += i;}
return res;
}
```

Figure 13: Code and specification sample

An obvious metrics in this context is thus to count the number of annotations that have been verified, the number of annotations for which a proof attempt failed, and the number of annotations that have been verified under the hypothesis that a yet-unproved annotation holds. In addition, this should be combined with the number of functions equipped with a contract. Indeed, by definition, only those functions are subject to verification through deductive proof-based tools. Thus, for the purpose of this kind of tools, the target of evaluation would be the set of formally specified functions.

While this metrics would provide a first evaluation of the amount of verification that has been done on a given code base, it is however not sufficient. In particular, if one puts too restrictive pre-conditions on a contract, it will become easier to prove the respective post-conditions, but at the expense of the callers. Conversely, a weak post-condition is easy to prove, but provide few guarantees. At worst, the following contract can be proved for any function:

```
/*@ requires \false;
    ensures \true;
*/
int f(int x);
```

Figure 14: Always-true contract

Thus, there is also a need to define some metrics assessing the quality of contracts. This point is a very new topic of investigation, and the proposals made here should not be taken as a definitive answer on this subject.

First, a measure of the quality of the pre-condition would be the proportion of execution branches that can be taken inside the function when starting from a context respecting the pre-condition over the total number of possible execution branches. This can be done by successively adding on each branch annotations that can only be proved if the branch is unreachable and check whether they are proved or not. For instance, in the following function, proving the assertion `dead` implies that the pre-condition is masking half of the function from the deductive tool, while being unable to prove `live` would indicate at least this branch is covered.

```
/*@ requires x > 0;
    ensures \result == 2;
*/
int f(int x) {
  if (x <= 0) {
    /*@ assert dead: \false; */
   return 1;
  } else {
   /*@ assert live: \false; */
   return 2;
  }
}
```

Figure 15: Metrics for pre-conditions

Second, for post-conditions, contracts could be required to have an `assigns` clause, and a metrics could be the proportion of memory locations that could be modified according to the `assigns` clause that appears in the post-conditions. Indeed, for such locations that are not mentioned, we do not have any information about the value that they can hold after the call. For instance, in the following code, our post-condition does not say anything about the value of `*p`, hence we do not know what the function might have done with it (note however that we still refer to the value of `*p` in the old state, but not to its actual value). From the point of view of a caller of `f`, the post-condition is incomplete, in the sense that it does not accurately describe the state of the program after a call to f. The purpose of the proposed metrics is to indicate which proportion of the function's footprint is described in its post-condition, as an indication of the quality of the contract.

```
/*@ requires \valid(p);
    assigns *p;
    ensures \result == \old(*p);
*/
int f(int *p) {
   int tmp = *p;
   *p = 42;
    return *p;
}
```

Figure 16: Post-conditions metrics

3.9 "CWE scoring of an alarm" Metric

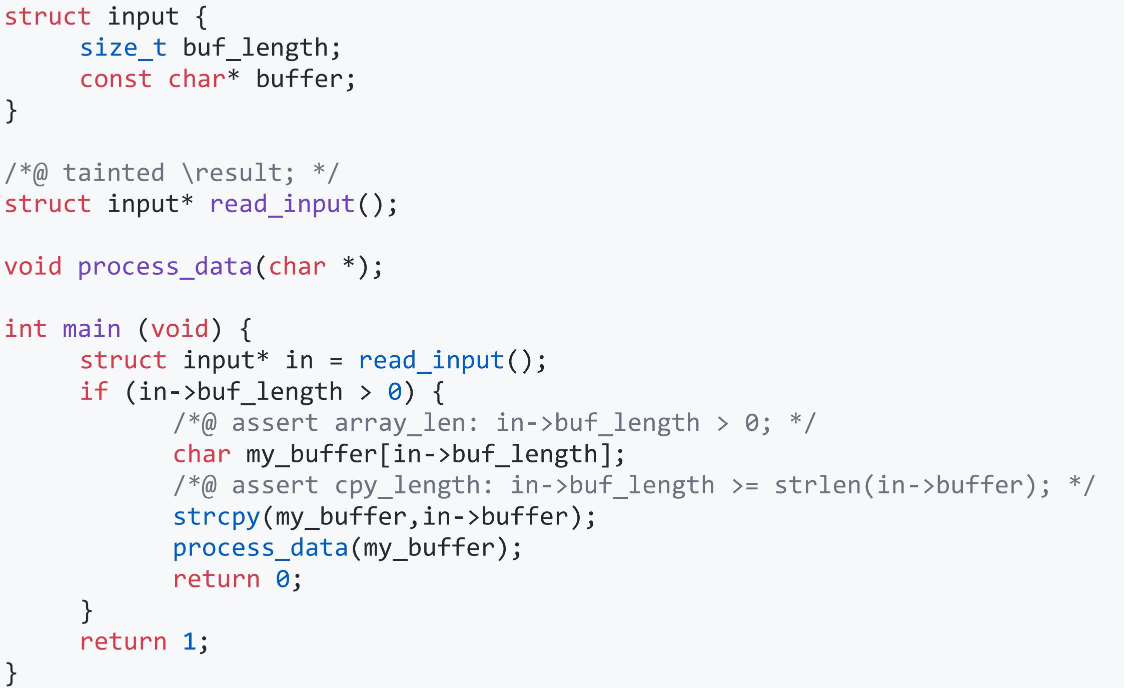
Security standards, such as CWE enumeration or CERT coding rules come with an evaluation of the potential severity of a given alarm. Hence, tools that can categorize the issues they report according to such standards will also be able to rank them according to these numbers. In particular, they could implement the Common Weakness Scoring System proposed by CWE (https://cwe.mitre.org/cwss/cwss_v1.0.1.html), or the Common Vulnerabilities Scoring System proposed by FIRST (https://www.first.org/cvss/). However, as noted in the document defining CWSS itself, it is very unlikely that an automated tool can compute an accurate score for all the components that are taken into account in the aggregated score, and some human intervention will be required to provide additional inputs.

Regarding Frama-C, no plug-in is currently reporting alarms according to CWE or CVE categorization, and it is not possible to obtain a CWSS scoring. Since a certain amount of user intervention is expected anyway, Frama-C MDR plug-in might be a suitable candidate to propose a template for CWSS and/or CVSS scoring of each alarm reported by EVA or WP, in order to help sorting these alarms according to their criticality. This template would then be filled by the user of MDR, in the same way as they currently provide additional information about each alarm in a free format. In particular, one of the major point of the current version of MDR is the possibility for the user to explain why an alarm raised EVA is in fact a false alarm, due to the abstractions made by the tool, and not a real vulnerability in the program under analysis. This is directly related to the "Finding Confidence" (FC) criterion of CWSS, and MDR users should thus easily be able to provide a suitable value for this criterion.

In addition, the taint analysis metrics proposed in the next section would provide valuable information for assessing the value of the "Internal Control Effectiveness" criterion, in the sense that the metrics is intended to show the distance between values from which at least one of them can be directly provided by an attacker and their use in operations that can lead to a runtime error.

3.10 “Criticality of an alarm” taint analysis Metrics

Taint analysis is an analysis technique that allows tracking where untrusted (“tainted”) input data can propagate to during the execution of a program. Based on this information, a useful metrics for evaluating the severity of an alarm is to compute the number of computational operations and/or tests over the tainted values that separate the values which trigger the alarm from tainted values (that must be assumed to be under the control of an attacker): the closer they are, the easier it will be for an attacker to craft a malicious vector of input data. This metrics assumes that the code has been annotated to indicate which functions read untrusted input and possibly which functions are able to sanitize such inputs.



```
struct input {
     size_t buf_length;
     const char* buffer;
}

/*@ tainted \result; */
struct input* read_input();

void process_data(char *);

int main (void) {
     struct input* in = read_input();
     if (in->buf_length > 0) {
          /*@ assert array_len: in->buf_length > 0; */
          char my_buffer[in->buf_length];
          /*@ assert cpy_length: in->buf_length >= strlen(in->buffer); */
          strcpy(my_buffer,in->buffer);
          process_data(my_buffer);
          return 0;
     }
     return 1;
}
```

Figure 17: Taint analysis example

If we take the example of Figure 17, where the function `read_input` is assumed to fill the structure it returns with data coming from an unknown source (meaning in particular that we cannot assume that `buf_length` contains the length of `buffer`), the first ACSL annotation is trivially true, as the value of `buf_length` is properly guarded by the condition just above. On the other hand, there are no checks of the consistency between the two fields of the structure, which means that the second ACSL annotation indicates a real alarm, which could directly be triggered by an attacker setting a `buf_length` lower than the actual length of `buffer`).

Frama-C currently has a plug-in dedicated to detection of information leak, which is based on techniques that are extremely similar to taint analysis, but no plug-in performs taint analysis as such.

3.11 “Statistics” Metric

The most straightforward metrics that can be computed on a program are purely syntactic information. They do not really inform about vulnerabilities per se, but can give a rough idea of the complexity of the code, hence of the amount of effort that will be required to analyze it. Many proposal have been made over the years to provide such metrics, such as cyclomatic complexity or Halstead’s complexity and can easily be implemented by the tools to provide a first overview of the code. The metrics plug-in of Frama-C already offers some of these metrics, both on the original (though post preprocessing) code and the normalized version on which the analyzers of the platform usually operate. The former should be privileged, as the normalization introduces constructions, notably `goto`s that are usually considered as making the code harder to understand (hence increasing the complexity) from a user point of view (as opposed to automated analysis plug-ins which are the primary users of the normalized code).

An example of the output provided by the Frama-C metrics plug-in is given in Figure 18 and Figure 19. The program under analysis is the bzip2 version that is analysed in the Open-Source Case Studies repository (https://github.com/Frama-C/open-source-case-studies/).

```
[metrics] Halstead metrics
    ================
    Total operators: 5494
    Distinct operators: 139
    Total_operands: 3185
    Distinct operands: 915
    Program length: 8679
    Vocabulary size: 1054
    Program volume: 87151.56
    Effort: 21083772.01
    Program level: 0.00
    Difficulty level: 241.92
    Time to implement: 1171320.67
    Bugs delivered: 25.44
```

```
    Global statistics (Halstead)
    ============================
    Operators
    ---------
      continue: 8
      const: 6
      typedef: 64
      goto: 34
…
```

Figure 18: Metrics over original code

```
 [metrics] Defined functions (44)
    ======================
     addFlagsFromEnvVar (2 calls); applySavedFileAttrToOutputFile (2
calls);
     applySavedTimeInfoToOutputFile (2 calls); cadvise (3 calls);
     cleanUpAndFail (8 calls); compress (2 calls); compressStream (1
call);
    ….
    Undefined functions (8)
    =======================
     BZ2_bzRead (2 calls); BZ2_bzReadClose (4 calls);
     BZ2_bzReadGetUnused (2 calls); BZ2_bzReadOpen (2 calls);
     BZ2_bzWrite (1 call); BZ2_bzWriteClose64 (2 calls);
     BZ2_bzWriteOpen (1 call); BZ2_bzlibVersion (2 calls);
    Potential entry points (1)
    ==========================
     main;
    Global metrics
    ==============
    Sloc = 1373
    Decision point = 352
    Global variables = 25
    If = 300
    Loop = 31
    Goto = 97
    Assignment = 389
    Exit point = 44
    Function = 52
    Function call = 421
    Pointer dereferencing = 95
    Cyclomatic complexity = 396
```

Figure 19: Metrics over normalized code

Chapter 4 Metrics in Frama-C

Currently, the main plug-ins in the Frama-C platform for providing metrics are the following:

- Report provides the status of each ACSL annotation (including the ones generated by other plug-ins, e.g. as the result of an alarm of EVA). Raw numbers of annotations that are valid, valid under hypothesis, unknown, or invalid can be obtained in a CSV table.
- MDR generates a markdown based template detailing the results of an analysis (mostly using EVA at this point). This template is meant to be completed by the user with additional information indicating e.g. why an alarm is spurious or whether a reported issue is mitigated through other parts of the system under analysis.
- Metrics provides syntactic metrics and, when launched after EVA, some basic coverage information (number of statements reached by EVA).

More information on these plug-in is available in D3.2. While the three plug-ins mentioned above will likely stay the preferred plug-ins for outputting relevant information to the user, some extensions to the analysis plug-ins themselves might be required. In particular, the proposals made to assess the quality of a specification in Section 3.8 would be better driven directly from the WP plug-in. Likewise, EVA has some internal information about the branches that it explores that is only available while the analysis is running and not kept in the analyzer’s state that can be queried by other plug-ins once the analysis has completed. Fine-grained branch coverage information as envisaged in Section 3.7 would thus imply modifications to EVA itself.

Taint analysis is not currently part of any plug-in that is included in Frama-C base distribution, hence the metrics of Section 3.10 will require new developments in order to be implemented. EVA and its associated plug-ins provide strong basic blocks for a taint analyzer, but some new annotations (for indicating which values are initially tainted, and what can be done to sanitize them) need to be designed, and written by the user of the analyzers.

Chapter 5 Summary and Conclusion

The following new metrics had been identified with the collaboration of different partners have been proposed and described in this document with design and implementation details on related tools:

Metrics name	Technical implementation	Type	Related plugin/tool
SecuRate	Fingerprint matching	Numeric value	SecuRate, EVA
CriticalDepth	Call stack depth	Numeric value	EVA
Liveness Metrics	Taint analysis	Numeric value	EVA
Size Definition Distance Metrics	Taint analysis	Numeric value	EVA
Dangling Pointers Persistence Metrics	Taint analysis	Numeric value	EVA
Cryptographic Secrets Persistence Metrics	Taint analysis	Numeric value	EVA
Static Analysis Coverage Metrics	Abstract Interpretation	Ratio	EVA
Quantitative assessment for deductive verification tools	Deductive Verification	Numeric value	WP
“CWE scoring of an alarm” Metric	Classification	Numeric value	Report/MDR
“Criticality of an alarm” taint analysis Metrics	Taint analysis / Control-flow assessment	Boolean / Numeric value	EVA
“Statistics” Metric	Collection of existing metrics	Set of numeric values	Metrics

Table 1: Table of new metrics

These metrics pose a relevant improvement in conducting security assessments and will enable future analysts and developers to determine and prioritize the most crucial vulnerabilities. Based description and specification of proposed metrics introduced in this deliverable, implementation activities will be performed in WP3. The integrated metrics will be demonstrated during further activities in the use-case evaluation in D4.6 [3] and quality tests in D4.5 [2].

Chapter 6 List of Abbreviations

Abbreviation	Translation
CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerabilities Scoring System
CWE	Common Weakness Enumeration
CWSS	Common Weakness Scoring System
EVA	Enhanced Value Analysis

Chapter 7 Bibliography

[1] Offutt, P. A. (2008). *Introduction to Software Testing.* New-York: Cambridge University Press.

[2] VESSEDIA DS-01-731453 / D4.5 report (unreleased): Quality tests & limits of VESSEDIA tools regarding security vulnerabilities detection

[3] VESSEDIA DS-01-731453 / D4.6 report (unreleased): Evaluation using the VESSEDIA use cases