POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Verifying MISRA C compliance in binary code - Discussion and practical examples

Supervisors

Prof. Alessandro SAVINO

Prof. Stefano DI CARLO

Dott. Franco OBERTI

Candidate

Matteo FANTOZZI

December 2024

Summary

Safety and security requirements of embedded systems have become more and more important over the years and various coding guidelines and standards have been release, with the object of regulating and assessing the safety and security of software running on these systems. But how to assess claimed compliance to a certain standard if the source code is not publicly available?

The thesis discusses the possibility of verifying several MISRA C coding guidelines in the absence of source code. Firstly, we analyze each mandatory MISRA rule in detail and assess whether it is possible to detect violations based on binary analysis alone. In addition, we present some prototype applications, based on binary analysis techniques, developed in order to detect these violations.

Acknowledgements

I'd obviously like to thank a few people. I'd like to thank Charlie Cryton, for doing the plot with me, and Jamie and Kevin for writing their parts. I would like to thank Michael Palin, for checking the scene numbers. I'd also like to thank John Comfort, Jonathan Benson, Roger Murray-Leech, Hazel Pessig, Glenn Palmer-Smith, Cynthia Kayla, Mariah Aitkin, Johnathan Aitkin, Lord Beaverbrook, Eleanor Roosevelt, Jack Cousteau, and his wife Mimi, Soren Kierkegaard, Gisela Werbezerk-Piffle, Sonny Liston and Hayden Jones, and her husband Pip, Gregor Mendel — the founder of the science of genetics — my tailor, Harriet Beacher-Stowe — author of Uncle Tom's Cabin — The London Symphony Orchestra Brass Section, The Leighton Orient Strikers, mother, Bismarck, The Royal Society for the Prevention of Birds, Sir Basil Smallpiece, St Francis of Assisi, Diana Ross and The Supremes, Earl Haig, Wily E. Coyote, Mother Teresa, Herb Alpert and his Tijuana Brass, Herman Goering, Dame Agatha Christie, the planet Saturn — and, of course, all of its rings — Joan Collins, The Publicity Department of Turkish Airways, the unknown soldier, Tammy Whinette, and last, but of course not least, God.

Table of Contents

List of Tables vii

List of Figures viii

Acronyms x

1 Introduction 1

1.1 Research goal 1

1.2 What is MISRA C 2

1.2.1 The C language and undefined behavior 2

1.2.2 Introducing MISRA C 2

1.3 MISRA C details and terminology 2

1.4 Why binary analysis 4

1.5 Existing work 4

2 Discussion of rules 6

2.1 Selection of rules 6

2.1.1 Rule 9.1 6

2.1.2 Rule 13.6 7

2.1.3 Rule 17.3 7

2.1.4 Rule 17.4 8

2.1.5 Rule 17.6 8

2.1.6 Rule 19.1 9

2.1.7 Rule 22.2 9

2.1.8 Rule 22.4 10

2.1.9 Rule 22.5 10

2.1.10 Rule 22.6 10

2.2 Summary 11

3 Practical analysis 12

3.1 Machine Learning approach 12

3.1.1 The dataset 12
3.1.2 The preprocessing 13
3.1.3 The classificator 14
3.1.4 Results 14
3.2 Algorithmic approach 18
3.2.1 Description 18
3.2.2 Results 18

4 Discussion of results and future directions 20
4.1 Machine Learning approach 20
4.1.1 Positives 20
4.1.2 Drawbacks 20
4.1.3 Improvements 21
4.2 Algorithmic approach 21
4.2.1 Positives 21
4.2.2 Drawbacks 21
4.2.3 Improvements 22

A Code snippets of MISRA rules 23

Bibliography 32

List of Tables

2.1 Summary of mandatory rules 11

3.1 Results of ML classification tests 16
3.2 Statistics 16
3.3 Results of *angr* tests 18

List of Figures

3.1 Flow chart of the Machine Learning process 15
3.2 Plot of training loss against epoch 17

A.1 Simple testcase for rule 9.1 23
A.2 Rule 9.1 compliant version binary 23
A.3 Rule 9.1 non-compliant version binary 24
A.4 Simple testcase for rule 17.4 24
A.5 Rule 17.4 compliant version binary 24
A.6 Rule 17.4 non-compliant version binary 25
A.7 Simple testcase for rule 19.1 25
A.8 Rule 19.1 compliant version binary 25
A.9 Rule 19.1 non-compliant version binary 26
A.10 Simple testcase for rule 22.2 26
A.11 Rule 22.2 compliant version binary 27
A.12 Rule 22.2 non-compliant version binary 27
A.13 Simple testcase for rule 22.4 28
A.14 Rule 22.4 compliant version binary 28
A.15 Rule 22.4 non-compliant version binary 29
A.16 Simple testcase for rule 22.5 29
A.17 Rule 22.5 compliant version binary 29
A.18 Rule 22.5 non-compliant version binary 30
A.19 Simple testcase for rule 22.6 30
A.20 Rule 22.6 compliant version binary 30
A.21 Rule 22.6 non-compliant version binary 31

Acronyms

MISRA
Motor Industry Software Reliability Association

WG14
Working Group 14

RNN
Recurrent Neaural Network

CWE
Common Weaknesses Enumeration

Chapter 1

Introduction

1.1 Research goal

The safety and security of embedded systems is of paramount importance as of today. Not only have these devices become drastically more numerous over the years, but the number and the consequence of their functions has steadily increased; that is to say, a mulfunction has very severe safety implications (e.g. embedded systems in vehicles, embedded systems in medical equipment).

To address these safety concerns, various guidelines and constraints have been placed on one of the most popular programming languages for embedded devices, the C language; among these is the MISRA C set of guidelines.

Such guidelines are a tool for developers to have a higher degree of confidence that the software they produce is safe and secure; but given that most software is closed source, how can the user or a third party verify compliance?

This being the premise, the object of this thesis is to investigate the possibility of checking MISRA C guidelines in the absence of the source code, based only on the machine code that is effectively run on devices.

The rest of this chapter provides more details on the specifics of the problem at hand, particularly the C langugage, MISRA C guidelines and the world of binary analysis. In Chapter 2 we analyze in detail those guidelines that we deem fit for analysis. In Chapter 3 we present the details of the analyses that we carried out. In Chapter 4 we discuss the results of these analyses and outline possible improvements and future directions.

1.2 What is MISRA C

The C language and undefined behavior

Despite being introduced over 50 years ago, C is still an important and widely used programming language; even more so in the field of embedded systems. The behavior of C code is regulated by the C Language Standard, which puts a set of constraints on C compilers. The C standard, however, does not cover every case and allows the compilers to "cut some corners", which allows them to be simpler to implement and to generate faster code; on the downside, this introduces *undefined behavior*. [1]
The official definition of undefined behavior implies that it's a programming error, however it is allowed and can lead to crashes, erratic behavior and is generally bad for applications that have safety and security requirements. [2]

Introducing MISRA C

MISRA (Motor Industry Software Reliability Association) is a consortium of manufacturers, component suppliers, engineering consultancies and academics, which «seeks to research and promote best practice in developing safety- and security-related electronic systems and other software-intensive applications», such as embedded control systems on cars. One of the results of this collaboration is MISRA C: a set of guidelines that aims at increasing the safety and security of a product that uses the C language. It does this by defining a subset of C that prevents the possibility of undefined behavior. Undefined behavior stems from the fact that the C language standard is not completely defined, and the behavior of certain bits of code is up to the compiler to decide. This creates code that may behave unexpectedly under certain conditions or on certain architectures. MISRA C helps developers avoid these unexpected behaviors, thus increasing the safety and security of a product.

MISRA C is not entirely about placing constraints on the source code, however; the adoption of MISRA C in a project must be accompanied by specific software development activities, in the official documentation. [3]. As prof. Bagnara explains, «a useful way to think about MISRA C and the processes around it is to consider them as an effective way of conducting a guided peer review to rule out most C language traps and pitfalls.» [2]

1.3 MISRA C details and terminology

MISRA C is, at its core, a list of detailed guidelines that apply not only to the C source code of a product, but to some stages of the development process as

well. All of the guidelines must be followed for a product to claim compliance with MISRA C, although some guidelines allow for *deviations*[1] (a formal explanation that justifies why it was deemed necessary to ignore a guideline).
Each guideline is either a *rule* or a *directive*:

- *Rules* provide a complete description of the requirement (that is, the limitations that allow for safe code); compliance can be checked through a static analysis tool.
- *Directives*, on the other hand, are often more broadly defined and cannot be checked through the source code alone; requirements, specifications, or design documents are needed to verify compliance.

Each guideline can belong to one of three categories:

- *Mandatory* guidelines are the most important, and must all be followed, always. Deviation is not allowed.
- *Required* guidelines must be followed, unless a deviation is provided.
- *Advisory* guidelines should be followed when possible. No formal deviation is required, although non-compliance cases still need to be documented.

Every rule can in turn be on of the following:

- *Decidable* rules are such that there exists (theoretically) a way to check compliance every time. This is a nice property, because it excludes the possibility of false positives or negatives.
- For *Undecidable* rules, on the other hand, this doesn't apply; therefore, there may be cases where it's impossible to statically verify if the rule has been violated or not. The best a tool can do, for such rules, is to adopt a broader approximation of the rule that guarantees no false negatives, at the cost of some false positives.

In addition, rules are also classified according to their scope:

- *Single Translation Unit* (STU) rules are such that the lowest amount of translation units (e.g a C file) required to detect a violation is one.
- *System* rules, on the other hand, are such that more than one translation unit is required in order to check it; possibly the entirety of the source code.

MISRA C:2012, which is the third version of the guidelines and the one this thesis is based on (before the addenda), contains 16 directives — of which 9 required and 7 advisory — and 143 rules — 10 mandatory, 101 required and 32 advisory. [3]

[1] All MISRA C core terminology used in this section will be marked by italics

1.4 Why binary analysis

As seen in the previous section, MISRA C rules apply to the C source code; they can be checked by using a combination of static analysis tools and manual inspection.

There are, however, cases where the source code is not available, but it is still desirable to assess the safety and security levels of the software: closed source projects and third-party libraries, for instance. The approach presented in this thesis applies in those cases where the "traditional" method just isn't viable.

There are apparent downsides to this approach. Source code and binary code are two radically different beasts. Aside from not being human-readable, binary code is just a sequence of instructions and data and carries no semantic information at all, compared to the source code it was generated from, or even the low-level assembly language used to represent it. This has two implications for us:

- not all MISRA C rules can be checked through binary code analysis. For some rules we can observe the behavior of the source code and infer some characteristic of the corresponding source code, and then judge whether that rule was violated or not; for other rules this is not possible, because the source code's characteristics that they regulate do not carry over to the binary code version.

- even in those cases where a rule can in fact be checked on the binary, since binary code does not have a 1:1 correspondence with source code, we will have to make assumptions and apply approximations in our analysis. This introduces the possibility of false positives and false negatives, which means the result of the analysis cannot be considered sound and complete.

At the end of section 2.1 is a table with a selection of MISRA rules that we believe can and cannot be checked by analysis binary code.

1.5 Existing work

When it comes to verifying MISRA C compliance on source code there is no shortage of proprietary and libre tools for the job; most of these are general-purpose static code analyzers that can be enabled to check MISRA C rules as well. On the contrary, relation between binary analysis and MISRA C has barely been considered; to the best of our knowledge, there are no academic resources on the subject.

The field of binary code analysis (BCA), however, is quite vast and its techniques can be useful for the problem at hand; these can generally be applied to different tasks, such as vulnerability detection, malware classification, reverse engineering

etc. Such techniques vary greatly and are usually classified in three categories: static analysis, dynamic analysis, and hybrid analysis.

[4] provides a good summary of binary analysis concepts, techniques and applications; it also introduced the powerful angr analysis engine.

There has also been a lot of effort to develop machine learning-based techniques for BCA [5]. Among these, we found the approach shown in [6] to be promising and we adopted it for one of our practical analyses. It is focused on the recognition of weaknesses in binary code at the function level, classified according to MITRE's CWE scheme.

Chapter 2

Discussion of rules

2.1 Selection of rules

We chose to limit our discussion to the 10 mandatory rules present in MISRA:2012. This provides a good starting point or our analysis, and we believe that the findings can be extended to the remaining rules without trouble. In view of what was said in section 1.4), the question we want to answer is: for which of these rules is it possible to detect a violation by looking just at binary code?

In order to determine that, for each rule, we wrote a handful of source code "testcases", with different data- and control-flow characteristics; each of these "testcases" in turn has two variants, one that violates that rule and one that doesn't. All of these variants are then compiled. If, when comparing the disassembled machine code that corresponds to the compliant and non-compliant variants, there are any meaningful differences (e.g. instructions in different order, different instructions or groups of instructions altogether etc.), analysis is deemed possible.

Here follows a list of the 10 mandatory rules, with each with a short description and the rationale behind them, and an explanation of whether we think analysis is possible or not. To better help understanding, Appendix A contains code examples, both source and binary, for the rules we determined to be verifiable.

Rule 9.1

Description This rule requires objects with automatic storage duration (that is to say, local variables allocated on the stack) to be initialized before being used.

Rationale Local variables in C are not automatically initialized, as opposed to objects with static storage duration; attempts to access uninitialized variables will result in using leftover memory content ("garbage") with unintended values, thus

leading to undefined behavior.

Detection The use of uninitialized variables is a common problem, and one that is not exclusive to C; one of the strategies that were devised to tackle it is 'definite assignment analysis', which uses and approximation that ensures no false negatives, at the cost of introducing some false positives. Another approach is to simply check that every stack location is written to before it is read; while simple, no guarantees can be made about false negatives and positives. While the concept of 'variable type' is lost on binary code, variables are still present simply as memory locations; we believe that by analyzing how memory locations are accessed, we can draw some conclusions about violations of this rule.

Rule 13.6

Description This rule prohibits using expressions that have side effects as arguments of the `sizeof()` operator. A few examples of side effects are accessing a volatile object, modifying an object, modifying a file, calling a function.

Rationale This rule exists because expressions inside the `sizeof()` operator are hardly ever evaluated at runtime; in the vast majority of cases the compiler itself does the math for us and replaces the expression with the correct value. Therefore, one cannot safely assume that the expression inside the operator will be executed (undefined behavior occurs).

Detection Unfortunately, we observed that in the majority of cases the `sizeof()` instruction translates to something as simple as storing a value in a general-purpose register; this is too generic an operation, therefore we can't reliably detect violations to this rule in binary analysis.

Rule 17.3

Description This rule prohibits implicit function declarations.

Rationale When the compiler can't find the declaration of a function, due to not including the proper header file, or due to omitting the declaration in the same source file, the compiler will generate a warning and assume that function's signature to have a return value of type `int` and parameters of type `int` as well. For most functions this assumption is obviously wrong; when those functions are called, undefined behavior occurs.

Detection Much like the previous rule, we did not observe meaningful differences in the disassembled code that can hint at a violation of the rule. This is because the typing system and the notion of 'return type' does not translate to binary code. Therefore, violations to this rule are not detectable in binary analysis.

Rule 17.4

Description This rule requires that:

- control do not reach the end of a non-void function without encountering a `return` statement;
- each `return` statement in a non-void function be followed by an expression[1].

Rationale As those who have encountered the "control reaches end of non-void function" GCC warning know, it's possible to compile code where some a function's control flow paths don't end with a return statement; if one of those paths is taken and the caller function uses the (nonexistent) return value, undefined behavior arises.

Detection The way C return values are handled by the processor is as follows: if the value's size is small enough, it will be stored in a specific register, right before the end of the function (e.g. in x86, that register is RAX); if it is a larger structure, it will be returned in the stack. The mechanics of return statements, therefore, leave a fair bit of traces in binary code; we believe these can be used to detect violations to this rule.

Rule 17.6

Description This rule prohibits the use of the `static` keyword between `[]` when declaring an array.

Rationale When declaring a function's parameters, the `static` keyword can be used to place a constraint on an array parameter that requires a minimum length; a call to such a function must provide an array with at least as many elements as indicated within the subscript operator `[]` in the function declaration. This allows for some compiler optimization, but if the programmer ignores the constraint and provides an array with fewer elements, undefined behavior occurs.

[1]`return` statements with no return value in non-void functions are only possible in C90

Detection The standard says that «a call to function may perform compile-time bounds checking and also permits optimizations such as prefetching»[2]. If such optimizations can be detected in binary code, and if they can be shown to be univocal with the use of the `static` keyword in an array declaration, then there would be a case for the detection of this rule. However, in the testcases we examined, we found no difference between versions that violate the rule and ones that don't; therefore we can conclude that violation may be detectable in some specific cases, but certainly not in all cases.

Rule 19.1

Description This rule prohibits copying or assigning an object to another, when the two objects' memory areas overlap. There are two exceptions however: either `memmove` is used, or the two objects overlap completely.

Rationale The rationale behind this rule is that, depending on how the memory areas overlap and how the copying process is implemented by the compiler (e.g. front-to-end, end-to-front), the result will be different; there is no way of knowing for sure what the result will be, and that is undefined behavior. This could happen when using C unions; it could also happen when copying using `memcpy`. The use of `memmove` is allowed because that function behaves as if using a temporary memory area for the copy, which is safe.

Detection We believe it's possible to detect violations similarly to rule 9.1. At the very least, usages of `memcpy` and `memmove` are easily detectable.

Rule 22.2

Description This rule prohibits double `free()`s, using `free()` on a block of memory that wasn't allocated with `malloc()` or `calloc()`, as well as using `realloc()` after a `free()`.

Rationale The rationale for this rule is self-evident: it bans incorrect usage of the dynamic memory management functions.

Detection If one can retrieve the names of function calls, it's possible to detect violations to this rule.

[2] https://en.cppreference.com/w/c/language/array

Rule 22.4

Description This rule prohibits writing to a read-only stream.

Rationale Writing data into a stream that is intended only for reading data is not described in the standard, therefore it results in undefined behavior. Streams, of course, are «a fairly abstract, high-level concept representing a communications channel to a file, device, or process»[3], described in C by the `FILE` data structure.

Detection Since it's possible to retrieve from the disassembled code the mode used to open a stream (`"r"`, `"w"` etc.), we believe it's possible to detect violations to this rule with similar techniques to the previous rules.

Rule 22.5

Description This rule prohibits dereferencing pointers to a `FILE` object, as well as copying the `FILE` objects directly (not the pointers).

Rationale The rationale for this rule is that one should only interact with `FILE` objects through the intended library functions and through pointers; manipulating the objects directly results in undefined behavior.

Detection This rule requires a double check:

- checking that the address returned by a `fopen()` is not accessed directly (a giveaway is its use in the base + offset memory access mode);
- checking that the address returned by a `fopen()` is not used as an argument for function calls other than the appropriate stream I/O library functions.

Rule 22.6

Description This rule prohibits any use of a pointer to a `FILE` object after the corresponding stream has been closed with `fclose()`.

Rationale Since the value of a `FILE` pointer is indeterminate after closing the stream, using it results in undefined behavior.

[3]https://www.gnu.org/software/libc/manual/html_node/I_002fO-on-Streams.html

Detection We believe violations to this rule can be detected similarly to the previous rule.

2.2 Summary

After this preliminary analysis, we expect to be able to detect violations of 7 of the 10 mandatory rules (the table below provides a concise summary). There is something important to note here: all 7 of these rules we deem detectable are *undecidable* rules, while the remaining 3 are all *decidable*. This is not entirely surprising, however, since decidability depends entirely on syntactical aspects of the source code; these aspects are no longer present in binary code, which makes us unable to check these rules. It's safe to assume that this applies to the rest of the rules in MISRA.

Rule no.	Detectable
9.1	✓
13.6	x
17.3	x
17.4	✓
17.6	x
19.1	✓
22.2	✓
22.4	✓
22.5	✓
22.6	✓

Table 2.1: Summary of mandatory rules

Chapter 3

Practical analysis

Now that we have closely examined the mandatory rules, we turn our attention to using known analysis techniques to develop tools that can perform automated analysis of binaries, in order to detect rule violations.

3.1 Machine Learning approach

We followed the method used by [6] to train a binary classificator model that can analyze some binary code and provide a yes/no answer to the question: is a specific rule being violated? We took rule 9.1 as our reference, but we are confident the process can be extended to the other rules.
In short, the model is trained using supervised learning, based on a Recurrent Neural Network that learns from the features of binary code that are provided by the `word2vec` algorithm. The following sections provide a summary of the process, from the data used to the classification results.

The dataset

Similarly to [6], we use the Juliet Test Suite for C/C++[1], version 1.3, part of the Software Assurance Reference Dataset published by NIST. The Juliet Test Suite for C/C++ is a collection of source code created with automatic tools by the NSA's Center for Assured Software as a benchmark for static analysis tools. The source code in this database is subdivided in 118 categories according to MITRE's Common Weakness Enumeration scheme; within each category are many *testcases*, bits of compilable code that exhibit the same weakness, but differ from each other

[1] available at https://github.com/arichardson/juliet-test-suite-c

in terms of control-flow and data types. Most testcases can be compiled so that a non-flawed version is produced, although some cannot (*bad-only* testcases).

The preprocessing

We closely follow the steps taken by [6] to implement feature extraction and feature encoding.

Build Firstly, we use the accompanying Python script to compile the testcases that belong to CWE-457 (Use of Uninitialized Variable) since it corresponds prefectly with the semantics of rule 9.1.

Decompile Then, we use the command line tool RetDec to decompile the binary code into LLVM Intermediate Representation. From this, we got 540 non-flawed, "good" code representations and 540 flawed ones.

Parse The Intermediate Representation code needs to be treated before it can be used for any sort of learning; so we built a custom parsing script in Python that performs 3 actions:

- filtering of functions that are not directly related to the weakness, as well as comments and global data, leaving only the functions that carry information about the weakness; it also filters functions with fewer than 300 tokens, and functions that exceed 1000 tokens.

- tokenization: splitting the remaining functions into individual syntactic tokens, adding End-of-Line tokens, removing whitespaces and the sort.

- name translation: replacing functions' and variables' names with generic names, so that there can be no correlation between them and the presence or absence of a weakness.

For each LLVM IR file, the parser transforms each relevant function into a list of strings, where each string represents a token. When parsing with a minimum function length of 300 tokens, we obtain a total of 1235 features: 921 of which are 'good' functions (no fault), and the remaining 314 'bad' functions (with a fault). The larger number of good functions is due to the fact that the majority of 'good' files contain a few variants of the non-flawed implementation, while the 'bad' files always contain only one.

Encode Next, the relevant functions extracted by the parses need to be transformed into a numerical format, so that a learning algorithm can be applied to them. For this, we apply the word2vec algorithm, using the `gensim` library, with its default parameters (that is, a vector size of 100, a window of 3, and sample rate of 0.001). First, we train the word2vec model on our corpus of tokens; then, we use the trained model to encode each token in each function, transforming it from a string to a 100-dimensional vector of floats.

With the processing stage complete, we obtained a corpus of numerical data that represents the essential elements of the binary code and is ready to be fed to a ML classificator. The entire corpus has the following dimensions: (1235, 100, 1000).

The classificator

The proposed machine learning algorithm follows the approach of [6], in that we train a binary classificator based on a RNN architecture. We use `pytorch` for the algorithm implementation, and `scikit-learn` for handling the data.

The dataset

The algorithm being a binary classificator, we only work with 2 labels. We give the non-flawed, "good" functions in our dataset the label '0', and '1' to the flawed functions, as the term 'positive' in this context refers to the presence of a flaw. When dividing the data between 'train' and 'test' sets, we apply a Stratified K-Fold Cross Validation scheme with 5 folds.

The model

The model is comprised of a Recurrent Neural Network with 4 layers, an input size of 100 (same as the word2vec encoding dimensions) and a hidden size of 64; the output of the RNN goes through a simple Linear readout layer with 2 output nodes. The classificator's prediction is considered to be the maximum value of the 2 output nodes. Training takes place in batches of 64, and stops at the 40th epoch.

Results

With an accuracy of 90% and a recall of 96.74% on the '0' class, the model is quite capable of recognising non-flawed instances. With a recall of 71% on the flawed class, we can deduce that the model introduces a lot of false positives.

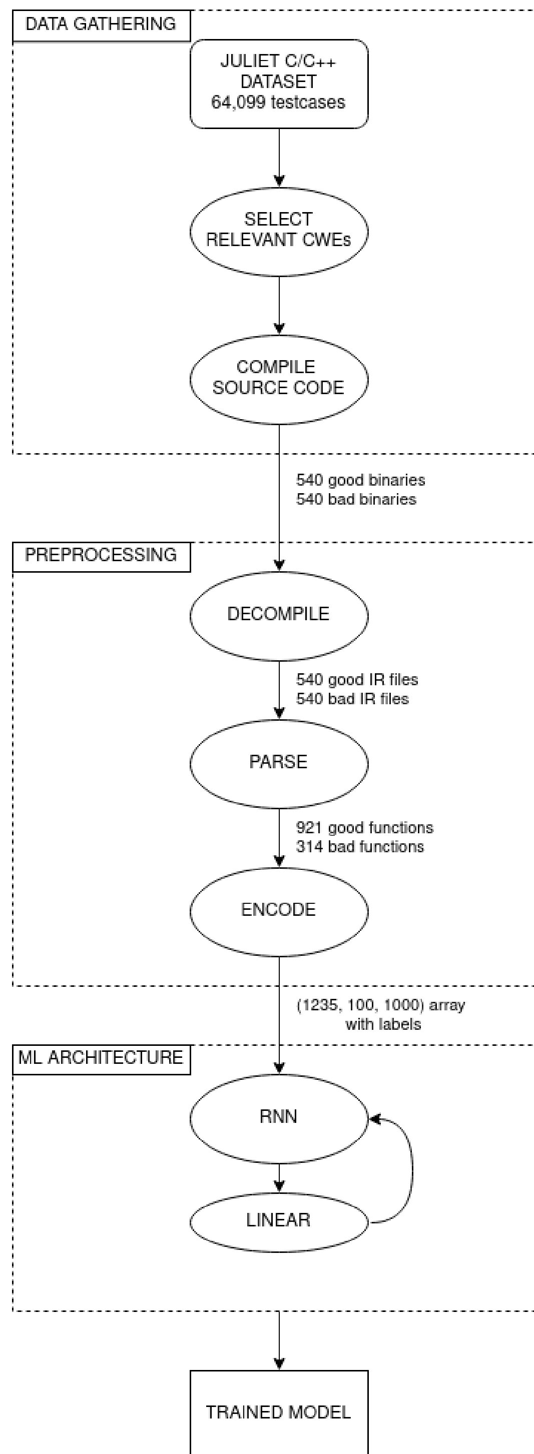

Figure 3.1: Flow chart of the Machine Learning process

Execution time was 1 hour, 42 minutes and 44 seconds with no GPU acceleration[2].

Fold	TN	FP	FN	TP	accuracy (baseline)
1	182	3	19	43	91.09% (73.68%)
2	172	12	18	45	87.85% (69.63%)
3	176	8	21	42	88.26% (71.25%)
4	179	5	15	48	91.90% (72.46%)
5	182	2	16	47	92.71% (73.68 %)
average	178.2	6	17.8	45	90.36% (72.14 %)

Table 3.1: Results of ML classification tests

Class ID	Precision	Recall	F1-score
0 (non-flawed)	90.91%	96.74%	93.73%
1 (flawed)	88.23%	71.65%	79.08%

Table 3.2: Statistics

[2]ASUS X756UXK (Intel Core i7-7500U CPU), 8 GB of RAM, 2 GB of swap RAM, SSD

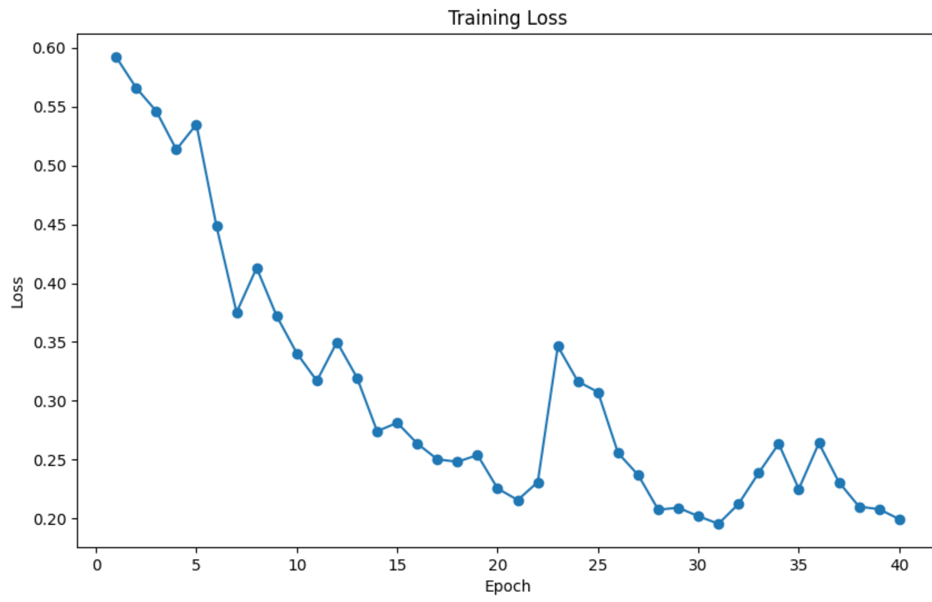

Figure 3.2: Plot of training loss against epoch

3.2 Algorithmic approach

We conducted a test using the tool *angr* to simulate the execution of binaries and detect violations of rule 9.1. *angr* [4] is a powerful software that incorporates various analysis tools in one package; its draws are support for several architectures, and built-in analysis techniques. It was designed to allow for easy reproducibility, and it's a useful tool for automating analyses.

Description

In order to detect violations to rule 9.1 automatically, we use a simple strategy that consists in tracking the reads and writes to the stack memory area for each function; we can detect a violation to the rule if a read over an untouched area of the stack occurs.

Each binary file that the algorithm analyzes is loaded to *angr* to obtain a `Project`, which is the main interface used to manipulate it. Through this, we get a list of the functions that were recognize din this binary; we initialize a collection of 'variable tables', where we will store the stack addresses touched by each function. Before starting the emulation, we set up 2 callback functions that are triggered when there are reads and writes to the stack; the 'write' function records on the appropriate 'variable table' the address written to, and the 'read' function checks whether the address being read is present in the 'variable table'; if not, it reports a violation.

Results

When testing this algorithm on the same dataset used for the previous test; we achieved an accuracy of 80%. However, we found that all the false positives we had were caused by the same specific condition (related to arrays and unrelated to the flaw); when that issue is isolated and accounted for, accuracy raises to 92.70%. Execution time was 28 minutes and 46 seconds[3].

TN	FP	FN	TP	accuracy (baseline)
1650	309	182	358	80.51% (78.34%)

Table 3.3: Results of *angr* tests

[3] ASUS X756UXK (Intel Core i7-7500U CPU), 8 GB of RAM, 2 GB of swap RAM, SSD

Algorithm 1 Pseudocode for rule 9.1 violation detection

```
▷ Initialization
initialize angr project
initialize variable tables

▷ Execution
start emulation
repeat
    if function call f() then
        t_f = init("f")                         ▷ initialize new table for function f
    end if

    if in function f, write operation to stack location a then
        add entry a to t_f                      ▷ write operation is recorded
    end if

    if in function f, read operation from stack location b then
        if t_f contains b then
            valid read
        else
            report violation
        end if
    end if

until emulation ends
```

Chapter 4

Discussion of results and future directions

4.1 Machine Learning approach

Positives

The Machine Learning approach to this analysis has some advantages:

- it is relatively easy to implement, and the analysis is easy to carry out;
- the same learning algorithm can be applied to target different rules, provided that a dataset of binary code can be supplied.
- the process can be easily extended to the other rules that we deemed fit for analysis in Chapter 2.

Drawbacks

- one of the weaknesses of this approach is that only those rules for which a substantial database can be supplied may be studied. The database examined in the examples we presented were assembled for different purposes, and they happened to fit well with the goal of our analysis.
- artificial code, as used in our examples, may come short of code found "in the wild". As [7] puts it: «Test cases are simpler than natural code. Some test cases are intentionally the simplest form of the flaw being tested. Even test cases which include control or data flow complexity are relatively simple compared to natural code.»

- since the learning process in non-deterministic, using a classification algorithm as shown in the example can't demonstrate the absence of faults. At best, it can be used to highlight critical code for further examination. [1]

Improvements

The first step in improving the results obtained in our test would be to improve the Machine Learning architecture itself; for instance, [6] observed that SRNN (Structural RNN) typically perform better, followed by LSTM (Long Short-Term Memory) Recurrent Neural Networks.

Secondly, the same Machine Learning process would be applied to the other rules for which we have a database and verify if accuracy stays high; using the same Juliet database, we would be able to analyze Rule 22.2 («A block of memory shall only be freed if it was allocated by means of a Standard Library function») by combining code from testcases of CWE-415 (Double Free) and CWE-590 (Free of Memory not on the Heap).

Thirdly, we would extend the current model to make it a multi-class classificator, capable of recognizing different types of flawed and non-flawed implementations. Alternatively, domain adaptation techniques can be investigated to extend the model to other rules.

4.2 Algorithmic approach

Positives

The algorithmic approach has, in principle, the possibility of not having any false negatives, if the correct approximation is chosen. This advantage, although hard to achieve, is crucial, as it ensures the soundness of the analysis.

Drawbacks

An algorithmic approach is generally harder to implement and requires very detailed knowledge of computer architectures, binary code execution and compilers.

The algorithmic approach is less scalable than the ML one; since, in general, every rule deals with different semantics, there is no one-size-fits-all approach. Every rule requires an ad-hoc algorithm that for every peculiarity of each rule.

[1]That might not be less of an issue as it appears to be. As [2] notes, the core of the compliance checking is not the pinpoint accuracy, but rather that violations are apparent and should not take long to be confirmed.

This is somewhat mitigated by the fact that many rules (e.g. all 22.* rules) are related and share the same semantics.

Improvements

Firstly, we wish to improve accuracy by eliminating the array issue that we identified, but couldn't get rid of due to a somewhat limited knowledge of the *angr* tool.

Secondly, we believe a more robust algorithm can be devised by employing concepts introduced by [8]; in particular, they perform concept assignment on binary code, utilizing formal analysis methods. They obtain good results in analyzing buffer overflow vulnerabilities, but are confident their approach can apply to other vulnerabilities as well.

Appendix A

Code snippets of MISRA rules

```
/**
 * The value of an object with automatic storage duration shall not be
 * read before it has been set
*/
int rule9_1_case1() {

    int a,b;

    #ifdef COMPLIANT

    a = 10;

    #endif /* COMPLIANT */

    b = a;

    return b;
}
```

Figure A.1: Simple testcase for rule 9.1

```
26: rule9_1_case1 ();
; var int64_t var_4h @ rbp-0x4
; var int64_t var_8h @ rbp-0x8
0x00001265      endbr64
0x00001269      push     rbp
0x0000126a      mov      rbp,     rsp
0x0000126d      mov      dword [var_8h],  0xa
0x00001274      mov      eax,     dword [var_8h]
0x00001277      mov      dword [var_4h],  eax
0x0000127a      mov      eax,     dword [var_4h]
0x0000127d      pop      rbp
0x0000127e      ret
```

Figure A.2: Rule 9.1 compliant version binary

```
19: rule9_1_case1 ();
; var int64_t var_4h @ rbp-0x4
; var int64_t var_8h @ rbp-0x8
0x00001285      endbr64
0x00001289      push    rbp
0x0000128a      mov     rbp,    rsp
0x0000128d      mov     eax,    dword [var_8h]
0x00001290      mov     dword [var_4h],   eax
0x00001293      mov     eax,    dword [var_4h]
0x00001296      pop     rbp
0x00001297      ret
```

Figure A.3: Rule 9.1 non-compliant version binary

```c
/**
 * All exit paths from a function with non-void return type shall have an
 * explicit return statement with an expression
*/
int rule17_4_case1(int a) {

    if(a < 0) {
        return -1;
    }

    #ifdef COMPLIANT

    return 1;

    #endif /* COMPLIANT */

}
```

Figure A.4: Simple testcase for rule 17.4

```
31: rule17_4_case1 (uint32_t arg1);
; arg uint32_t arg1 @ rdi
; var uint32_t var_4h @ rbp-0x4
0x00001354      endbr64
0x00001358      push    rbp
0x00001359      mov     rbp,    rsp
0x0000135c      mov     dword [var_4h],   edi          ; arg1
0x0000135f      cmp     dword [var_4h],   0
0x00001363      jns     0x136c                         ; unlikely
0x00001365      mov     eax,    0xffffffff             ; -1
0x0000136a      jmp     0x1371
0x0000136c      mov     eax,    1
0x00001371      pop     rbp
0x00001372      ret
```

Figure A.5: Rule 17.4 compliant version binary

```
26: rule17_4_case1 (uint32_t arg1);
; arg uint32_t arg1 @ rdi
; var uint32_t var_4h @ rbp-0x4
0x00001379      endbr64
0x0000137d      push     rbp
0x0000137e      mov      rbp,     rsp
0x00001381      mov      dword [var_4h],   edi           ; arg1
0x00001384      cmp      dword [var_4h],   0
0x00001388      jns      0x1391
0x0000138a      mov      eax,     0xffffffff             ; -1
0x0000138f      jmp      0x1391
0x00001391      pop      rbp
0x00001392      ret
```

Figure A.6: Rule 17.4 non-compliant version binary

```
void rule19_1_case2() {

    char *dest = calloc(12, sizeof(char));
    strncpy(dest, "abcdefghijk", 11);
    char *src = dest + 8; // "ijk"

    #ifndef COMPLIANT

    memcpy(dest, src, 3);     // memcpy doesn't account for overlap;

    #else

    memmove(dest, src, 3);

    #endif /* COMPLIANT */
}
```

Figure A.7: Simple testcase for rule 19.1

```
97: rule19_1_case2 ();
; var void *s2 @ rbp-0x8
; var void *s1 @ rbp-0x10
0x0000160c      endbr64
0x00001610      push     rbp
0x00001611      mov      rbp,     rsp
0x00001614      sub      rsp,     0x10
0x00001618      mov      esi,     1                      ; size_t size
0x0000161d      mov      edi,     0xc                    ; size_t nmeb
0x00001622      call     calloc                          ; sym.imp.calloc
; void *calloc(0x00000000, 0x02464c45)
0x00001627      mov      qword [s1],   rax
0x0000162b      mov      rax,     qword [s1]
0x0000162f      movabs   rdx,     0x6867666564636261     ; 'abcdefgh'
0x00001639      mov      qword [rax],   rdx
0x0000163c      mov      word [rax + 8],   0x6a69        ; 'ij'
0x00001642      mov      byte [rax + 0xa],   0x6b        ; 'k'
0x00001646      mov      rax,     qword [s1]
0x0000164a      add      rax,     8
0x0000164e      mov      qword [s2],   rax
0x00001652      mov      rcx,     qword [s2]
0x00001656      mov      rax,     qword [s1]
0x0000165a      mov      edx,     3                      ; size_t n
0x0000165f      mov      rsi,     rcx                    ; const void *s2
0x00001662      mov      rdi,     rax                    ; void *s1
0x00001665      call     memmove                         ; sym.imp.memmove
; void *memmove(-1, -1, 0x01010246)
0x0000166a      nop
0x0000166b      leave
0x0000166c      ret
```

Figure A.8: Rule 19.1 compliant version binary

```
97: rule19_1_case2 ();
; var void *s2 @ rbp-0x8
; var void *s1 @ rbp-0x10
0x00001601      endbr64
0x00001605      push     rbp
0x00001606      mov      rbp,    rsp
0x00001609      sub      rsp,    0x10
0x0000160d      mov      esi,    1                                  ; size_t size
0x00001612      mov      edi,    0xc                                ; size_t nmeb
0x00001617      call     calloc                                     ; sym.imp.calloc
0x0000161c      mov      qword [s1],  rax
0x00001620      mov      rax,    qword [s1]
0x00001624      movabs   rdx,    0x6867666564636261                 ; 'abcdefgh'
0x0000162e      mov      qword [rax],  rdx
0x00001631      mov      word [rax + 8],  0x6a69                    ; 'ij'
0x00001637      mov      byte [rax + 0xa],  0x6b                    ; 'k'
0x0000163b      mov      rax,    qword [s1]
0x0000163f      add      rax,    8
0x00001643      mov      qword [s2],  rax
0x00001647      mov      rcx,    qword [s2]
0x0000164b      mov      rax,    qword [s1]
0x0000164f      mov      edx,    3                                  ; size_t n
0x00001654      mov      rsi,    rcx                                ; const void *s2
0x00001657      mov      rdi,    rax                                ; void *s1
0x0000165a      call     memcpy                                     ; sym.imp.memcpy
0x0000165f      nop
0x00001660      leave
0x00001661      ret
```

Figure A.9: Rule 19.1 non-compliant version binary

```
void rule22_2_case3(int a) {        // free after free

    char *ptr = malloc(256);

        if(a>0) {
            ptr[0] = 'b';
            ptr[1] = '\000';
            #ifndef COMPLIANT

        free(ptr);

        #endif  /* COMPLIANT */
        }


    free(ptr);
}
```

Figure A.10: Simple testcase for rule 22.2

```
68: rule22_2_case3 (signed int64_t arg1);
; arg signed int64_t arg1 @ rdi
; var void *ptr @ rbp-0x8
; var signed int64_t var_14h @ rbp-0x14
0x00001735      endbr64
0x00001739      push    rbp
0x0000173a      mov     rbp,    rsp
0x0000173d      sub     rsp,    0x20
0x00001741      mov     dword [var_14h],  edi            ; arg1
0x00001744      mov     edi,    0x100                    ; size_t size
0x00001749      call    malloc                           ; sym.imp.malloc
;  void *malloc(0x00001000)
0x0000174e      mov     qword [ptr],  rax
0x00001752      cmp     dword [var_14h],  0
0x00001756      jle     0x176a                           ; likely
0x00001758      mov     rax,    qword [ptr]
0x0000175c      mov     byte [rax],  0x62                ; 'b'
0x0000175f      mov     rax,    qword [ptr]
0x00001763      add     rax,    1
0x00001767      mov     byte [rax],  0
0x0000176a      mov     rax,    qword [ptr]
0x0000176e      mov     rdi,    rax                      ; void *ptr
0x00001771      call    section..plt.sec                 ; sym.imp.free ;
; void free(-1)
0x00001776      nop
0x00001777      leave
0x00001778      ret
```

Figure A.11: Rule 22.2 compliant version binary

```
80: rule22_2_case3 (signed int64_t arg1);
; arg signed int64_t arg1 @ rdi
; var void *ptr @ rbp-0x8
; var signed int64_t var_14h @ rbp-0x14
0x0000172c      endbr64
0x00001730      push    rbp
0x00001731      mov     rbp,    rsp
0x00001734      sub     rsp,    0x20
0x00001738      mov     dword [var_14h],  edi            ; arg1
0x0000173b      mov     edi,    0x100                    ; size_t size
0x00001740      call    malloc                           ; sym.imp.malloc
0x00001745      mov     qword [ptr],  rax
0x00001749      cmp     dword [var_14h],  0
0x0000174d      jle     0x176d
0x0000174f      mov     rax,    qword [ptr]
0x00001753      mov     byte [rax],  0x62                ; 'b'
0x00001756      mov     rax,    qword [ptr]
0x0000175a      add     rax,    1
0x0000175e      mov     byte [rax],  0
0x00001761      mov     rax,    qword [ptr]
0x00001765      mov     rdi,    rax                      ; void *ptr
0x00001768      call    section..plt.sec                 ; sym.imp.free ;
0x0000176d      mov     rax,    qword [ptr]
0x00001771      mov     rdi,    rax                      ; void *ptr
0x00001774      call    section..plt.sec                 ; sym.imp.free ;
0x00001779      nop
0x0000177a      leave
0x0000177b      ret
```

Figure A.12: Rule 22.2 non-compliant version binary

```c
/**
* There shall be no attempt to write to a stream which has been opened as read-only
*/
void rule22_4() {

    FILE *fp = fopen ( "foo", "r" );
    int d = 0;

    #ifndef COMPLIANT

    fprintf ( fp, "What happens now?" );

    #else

    fscanf(fp, "%d", &d);

    #endif  /* COMPLIANT */
} // cppcheck-suppress resourceLeak
```

Figure A.13: Simple testcase for rule 22.4

```
117: rule22_4 ();
; var int64_t canary @ rbp-0x8
; var file*stream @ rbp-0x10
; var int64_t var_14h @ rbp-0x14
0x000017b7      endbr64
0x000017bb      push     rbp
0x000017bc      mov      rbp,    rsp
0x000017bf      sub      rsp,    0x20
0x000017c3      mov      rax,    qword fs:[0x28]              ; elf_shdr
0x000017cc      mov      qword [canary],  rax
0x000017d0      xor      eax,    eax
0x000017d2      lea      rax,    [0x00002008]                 ; "r"
0x000017d9      mov      rsi,    rax                          ; const char *mode ; "r"
0x000017dc      lea      rax,    [0x0000200a]                 ; "foo"
0x000017e3      mov      rdi,    rax                          ; const char *filename ; "foo"
0x000017e6      call     fopen                                ; sym.imp.fopen ; file*fopen(cor
; file*fopen("foo", "r")
0x000017eb      mov      qword [stream],  rax
0x000017ef      mov      dword [var_14h],  0
0x000017f6      lea      rdx,    [var_14h]                    ;  ...
0x000017fa      mov      rax,    qword [stream]
0x000017fe      lea      rcx,    [0x0000200e]                 ; "%d"
0x00001805      mov      rsi,    rcx                          ; const char *format ; "%d"
0x00001808      mov      rdi,    rax                          ; FILE *stream
0x0000180b      mov      eax,    0
0x00001810      call     __isoc99_fscanf                      ; sym.imp.__isoc99_fscanf ; int
; int fscanf(?, "%d", ?)
0x00001815      nop
0x00001816      mov      rax,    qword [canary]
0x0000181a      sub      rax,    qword fs:[0x28]
0x00001823      je       0x182a                               ; unlikely
0x00001825      call     __stack_chk_fail                     ; sym.imp.__stack_chk_fail ; voi
; void __stack_chk_fail(void)
0x0000182a      leave
0x0000182b      ret
```

Figure A.14: Rule 22.4 compliant version binary

```
83: rule22_4 ();
; var file*stream @ rbp-0x8
; var int64_t var_ch @ rbp-0xc
0x000017ba      endbr64
0x000017be      push     rbp
0x000017bf      mov      rbp,    rsp
0x000017c2      sub      rsp,    0x10
0x000017c6      lea      rax,    [0x00002008]
0x000017cd      mov      rsi,    rax                     ; const char *mode
0x000017d0      lea      rax,    [0x0000200a]
0x000017d7      mov      rdi,    rax                     ; const char *filename
0x000017da      call     fopen                           ; sym.imp.fopen ; file*
0x000017df      mov      qword [stream],  rax
0x000017e3      mov      dword [var_ch],  0
0x000017ea      mov      rax,    qword [stream]
0x000017ee      mov      rcx,    rax                     ; FILE *stream
0x000017f1      mov      edx,    0x11                    ; size_t nitems
0x000017f6      mov      esi,    1                       ; size_t size
0x000017fb      lea      rax,    str.What_happens_now_   ; 0x200e
0x00001802      mov      rdi,    rax                     ; const void *ptr
0x00001805      call     fwrite                          ; sym.imp.fwrite ; size
0x0000180a      nop
0x0000180b      leave
0x0000180c      ret
```

Figure A.15: Rule 22.4 non-compliant version binary

```
void rule22_5_case2() {            // indirect dereference by copy
    FILE *file1 = fopen ( "foo", "r" );
    FILE *file2 = fopen ( "foo", "r" );


    #ifndef COMPLIANT
        memcpy(file2, file1, sizeof(FILE));             // cppcheck-
    #else
        file2 = file1;
    #endif /* COMPLIANT */
} // cppcheck-suppress resourceLeak
```

Figure A.16: Simple testcase for rule 22.5

```
81: rule22_5_case2 ();
; var file*var_8h @ rbp-0x8
; var file*var_10h @ rbp-0x10
0x0000188f      endbr64
0x00001893      push     rbp
0x00001894      mov      rbp,    rsp
0x00001897      sub      rsp,    0x10
0x0000189b      lea      rax,    [0x00002008]
0x000018a2      mov      rsi,    rax                 ; const char *mode
0x000018a5      lea      rax,    [0x0000200a]
0x000018ac      mov      rdi,    rax                 ; const char *filename
0x000018af      call     fopen                       ; sym.imp.fopen ; file*fopen(
0x000018b4      mov      qword [var_10h],  rax
0x000018b8      lea      rax,    [0x00002008]
0x000018bf      mov      rsi,    rax                 ; const char *mode
0x000018c2      lea      rax,    [0x0000200a]
0x000018c9      mov      rdi,    rax                 ; const char *filename
0x000018cc      call     fopen                       ; sym.imp.fopen ; file*fopen(
0x000018d1      mov      qword [var_8h],  rax
0x000018d5      mov      rax,    qword [var_10h]
0x000018d9      mov      qword [var_8h],  rax
0x000018dd      nop
0x000018de      leave
0x000018df      ret
```

Figure A.17: Rule 22.5 compliant version binary

```
97: rule22_5_case2 ();
; var file*s1 @ rbp-0x8
; var file*s2 @ rbp-0x10
0x00001990      endbr64
0x00001994      push    rbp
0x00001995      mov     rbp,    rsp
0x00001998      sub     rsp,    0x10
0x0000199c      lea     rax,    [0x00002008]
0x000019a3      mov     rsi,    rax                     ; const char *mode
0x000019a6      lea     rax,    [0x0000200a]
0x000019ad      mov     rdi,    rax                     ; const char *filename
0x000019b0      call    fopen                           ; sym.imp.fopen ; file*fopen(
0x000019b5      mov     qword [s2],  rax
0x000019b9      lea     rax,    [0x00002008]
0x000019c0      mov     rsi,    rax                     ; const char *mode
0x000019c3      lea     rax,    [0x0000200a]
0x000019ca      mov     rdi,    rax                     ; const char *filename
0x000019cd      call    fopen                           ; sym.imp.fopen ; file*fopen(
0x000019d2      mov     qword [s1],  rax
0x000019d6      mov     rcx,    qword [s2]
0x000019da      mov     rax,    qword [s1]
0x000019de      mov     edx,    0xd8                    ; size_t n
0x000019e3      mov     rsi,    rcx                     ; const void *s2
0x000019e6      mov     rdi,    rax                     ; void *s1
0x000019e9      call    memcpy                          ; sym.imp.memcpy ; void *memc
0x000019ee      nop
0x000019ef      leave
0x000019f0      ret
```

Figure A.18: Rule 22.5 non-compliant version binary

```
/**
 * The value of a pointer to a FILE shall not be used after the associated stream has been closed
*/
void rule22_6() {
    FILE *fp;

    fp = fopen ( "tmp", "w" );
    fclose ( fp );

    #ifndef COMPLIANT

    fprintf (fp, "%s", "illegal");

    #endif /* COMPLIANT */
} // cppcheck-suppress resourceLeak
```

Figure A.19: Simple testcase for rule 22.6

```
56: rule22_6 ();
; var file*stream @ rbp-0x8
0x0000193a      endbr64
0x0000193e      push    rbp
0x0000193f      mov     rbp,    rsp
0x00001942      sub     rsp,    0x10
0x00001946      lea     rax,    [0x00002011]
0x0000194d      mov     rsi,    rax                     ; const char *mode
0x00001950      lea     rax,    [0x00002013]
0x00001957      mov     rdi,    rax                     ; const char *filename
0x0000195a      call    fopen                           ; sym.imp.fopen ; file*f
0x0000195f      mov     qword [stream],  rax
0x00001963      mov     rax,    qword [stream]
0x00001967      mov     rdi,    rax                     ; FILE *stream
0x0000196a      call    fclose                          ; sym.imp.fclose ; int f
0x0000196f      nop
0x00001970      leave
0x00001971      ret
```

Figure A.20: Rule 22.6 compliant version binary

```
88: rule22_6 ();
; var file*stream @ rbp-0x8
0x00001a55      endbr64
0x00001a59      push     rbp
0x00001a5a      mov      rbp,    rsp
0x00001a5d      sub      rsp,    0x10
0x00001a61      lea      rax,    [0x00002020]
0x00001a68      mov      rsi,    rax                      ; const char *mode
0x00001a6b      lea      rax,    [0x00002022]
0x00001a72      mov      rdi,    rax                      ; const char *filename
0x00001a75      call     fopen                            ; sym.imp.fopen ; file*f
0x00001a7a      mov      qword [stream],  rax
0x00001a7e      mov      rax,    qword [stream]
0x00001a82      mov      rdi,    rax                      ; FILE *stream
0x00001a85      call     fclose                           ; sym.imp.fclose ; int f
0x00001a8a      mov      rax,    qword [stream]
0x00001a8e      mov      rcx,    rax                      ; FILE *stream
0x00001a91      mov      edx,    7                        ; size_t nitems
0x00001a96      mov      esi,    1                        ; size_t size
0x00001a9b      lea      rax,    str.illegal              ; 0x2026
0x00001aa2      mov      rdi,    rax                      ; const void *ptr
0x00001aa5      call     fwrite                           ; sym.imp.fwrite ; size_
0x00001aaa      nop
0x00001aab      leave
```

Figure A.21: Rule 22.6 non-compliant version binary

Bibliography

[1] WG14. *ISO/IEC 9899:TC3 (committee draft)*. Tech. rep. ISO, 2007 (cit. on p. 2).

[2] *SAS2018 - The Misra C Coding Standard and its Role in the Development (by Roberto Bagnara)*. 2018. URL: `https://www.youtube.com/watch?v=LCZotsYizRI&t=2921s` (cit. on pp. 2, 21).

[3] Banham et al. *MISRA C:2012 Guidelines for the use of the C language in critical systems*. Tech. rep. MISRA Consortium, Mar. 2013 (cit. on pp. 2, 3).

[4] Yan Shoshitaishvili et al. «SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis». In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016, pp. 138–157. DOI: `10.1109/SP.2016.17` (cit. on pp. 5, 18).

[5] Hongfa Xue, Shaowen Sun, Guru Venkataramani, and Tian Lan. «Machine Learning-Based Analysis of Program Binaries: A Comprehensive Study». In: *IEEE Access* 7 (2019), pp. 65889–65912. DOI: `10.1109/ACCESS.2019.2917668` (cit. on p. 5).

[6] Andreas Schaad and Dominik Binder. *Deep-Learning-based Vulnerability Detection in Binary Executables*. 2022. arXiv: `2212.01254 [cs.CR]`. URL: `https://arxiv.org/abs/2212.01254` (cit. on pp. 5, 12–14, 21).

[7] *Juliet Test Suite v1.2 for C/C++ User Guide*. Center for Assured Software, National Security Agency. Dec. 2012 (cit. on p. 20).

[8] Z.D. Sisco and A.R. Bryant. «A Semantics-Based Approach to Concept Assignment in Assembly Code». In: *Proceedings of the 12th International Conference on Cyber Warfare and Security*. 2017 (cit. on p. 22).