

---

Análisis de código e instrumentación de los  
envíos para jueces en línea  
Code analysis and instrumentation of  
submissions for online judges

---



Trabajo de Fin de Grado

Curso 2023–2024

**Autores**

Luis Esteban Velasco

Juan Trillo Carreras

Rodrigo Burgos Sosa

**Director**

Rubén Rafael Rubio Cuéllar

Grado en Ingeniería Informática

Facultad de Informática

Universidad Complutense de Madrid



Análisis de código e instrumentación de los  
envíos para jueces en línea  
Code analysis and instrumentation of  
submissions for online judges

Trabajo de Fin de Grado en Ingeniería Informática

**Autores**

Luis Esteban Velasco  
Juan Trillo Carreras  
Rodrigo Burgos Sosa

**Director**

Rubén Rafael Rubio Cuéllar

**Convocatoria:** *Junio 2024*

Grado en Ingeniería Informática  
Facultad de Informática  
Universidad Complutense de Madrid

27 de mayo de 2024



# Agradecimientos

A nuestras familias por su apoyo, al equipo de DOMjudge y finalmente a Rubén por su entusiasmo y dedicación como tutor de este trabajo de fin de grado.



# Resumen

## Análisis de código e instrumentación de los envíos para jueces en línea

La práctica autónoma es fundamental en el aprendizaje de la programación y en los diversos contextos donde se enseña esta disciplina los correctores automáticos de problemas son muy utilizados. Los llamados jueces en línea ofrecen un repertorio de problemas y permiten enviar a los usuarios sus soluciones para que sean evaluadas en el momento. Para evaluarlas, el juez recibe el código del usuario, lo compila y lo ejecuta sobre distintos ejemplos para después comprobar que los resultados coincidan con los esperados. El resultado de esa evaluación para el estudiante se limita habitualmente a un veredicto que indica sin detalles si la solución es correcta o no, tarda demasiado o falla al ejecutarse. El profesor, en cambio, ve las diferencias en bruto entre la salida del programa y la salida esperada. En los concursos de programación, esta información es más que suficiente, pero una retroalimentación más instructiva y significativa sería de gran ayuda para el estudiante que está aprendiendo a programar.

El objetivo de este trabajo es aplicar técnicas de análisis de estático e instrumentación en los jueces automáticos para proporcionar retroalimentación más significativa a estudiantes y profesores.

Se ha estudiado la efectividad de un pequeño conjunto de herramientas sobre envíos reales de años anteriores. Las conclusiones de este análisis, interesantes por sí mismas, nos han servido para decidir qué comprobaciones son más relevantes para su implementación al juez. Hemos extendido el juez automático DOMjudge, ampliamente utilizado en la facultad de Informática de la UCM y en otras universidades, para integrar estas comprobaciones y mostrar los resultados en la retroalimentación a los estudiantes. Esto ha requerido modificaciones en el código y en el proceso de instalación del juez automático, y el desarrollo de herramientas para procesar la información de los analizadores.

## Palabras clave

Juez en línea, Análisis estático, Instrumentación, C++, DOMjudge





# Abstract

## Code analysis and instrumentation of submissions for online judges

Autonomous practice is fundamental in learning programming and, in the various contexts where programming is taught, automatic problem checkers are extensively used. The so-called online judges offer a repertoire of problems and allow users to submit their solutions for on-the-spot evaluation. The judge receives the code from the user, compiles it, and runs it on different test inputs, then checks that the results match the expected ones. The result of this evaluation for the student is usually limited to a verdict that indicates without details whether the solution is correct or not, takes too long or fails to run. The teacher, on the other hand, sees the raw differences between the program output and the expected output. In programming competitions, this information is more than enough, but more instructive and meaningful feedback would be of great help to the student learning to code.

The aim of this work is to apply static code analysis techniques and instrumentation on automatic judges to provide more meaningful feedback to students and teachers.

The effectiveness of a small set of tools has been studied on actual submissions from previous years. The conclusions of this analysis, interesting in their own, have helped us decide which checks are most relevant to implement on the judge. We have extended the online judge DOMjudge, (widely used in the Faculty of Computer Science at UCM and other universities) to integrate these checks and include the results in the feedback. This has required modifications to the code and the setup process of the automatic judge, and the development of tools to process the information from those tools.

## Keywords

Online Judge, Static analysis, Instrumentation, C++, DOMjudge



# Contents

<b>1. Introduction</b>	<b>1</b>
1.1. Motivation . . . . .	2
1.2. Goals . . . . .	3
1.3. Work Plan . . . . .	4
1.4. Organization of this thesis . . . . .	5
1.5. Source code and data repositories . . . . .	5
<b>2. State of the art</b>	<b>7</b>
2.1. Online judges . . . . .	7
2.1.1. DOMjudge . . . . .	8
2.2. Static analysis . . . . .	10
2.2.1. Clang static analyzer and Clang-Tidy . . . . .	10
2.2.2. Cppcheck . . . . .	11
2.3. Instrumentation . . . . .	11
2.3.1. AddressSanitizer . . . . .	12
2.3.2. UndefinedBehaviorSanitizer . . . . .	12
<b>3. Offline analysis on real student submissions</b>	<b>13</b>
3.1. Context of the course . . . . .	13
3.2. Data extraction and manipulation . . . . .	13
3.3. Code analysis results . . . . .	14
3.3.1. Clang-Tidy . . . . .	15
3.3.2. Cppcheck . . . . .	17
3.4. Instrumentation results . . . . .	18
3.4.1. Memory errors by ASan . . . . .	18
3.4.2. Undefined behavior by UBSan . . . . .	19
3.5. Conclusions . . . . .	20
<b>4. Integration of static and runtime analysis in DOMjudge</b>	<b>21</b>
4.1. More on the DOMjudge architecture . . . . .	22
4.1.1. The structure . . . . .	22
4.1.2. Submission environment . . . . .	23

4.1.3.	Submission process . . . . .	23
4.1.4.	More on files . . . . .	24
4.2.	Running the tools . . . . .	26
4.2.1.	Implementation alternatives . . . . .	30
4.3.	Providing feedback . . . . .	31
4.3.1.	Obtaining the number of tries . . . . .	32
4.3.2.	Generating the message . . . . .	35
4.4.	Preparations and Setup . . . . .	35
4.4.1.	Initializing the containers . . . . .	36
4.4.2.	Building new images . . . . .	37
4.4.3.	Automating the process . . . . .	38
<b>5.</b>	<b>Real use case scenario</b>	<b>41</b>
5.1.	Showcase . . . . .	41
5.2.	Realistic Case Scenario . . . . .	48
<b>6.</b>	<b>Conclusions and Future Work</b>	<b>53</b>
6.1.	Discussion and conclusions . . . . .	53
6.2.	Future work . . . . .	54
<b>7.</b>	<b>Personal Contributions</b>	<b>55</b>
7.1.	Rodrigo Burgos Sosa . . . . .	55
7.2.	Juan Trillo Carreras . . . . .	57
7.3.	Luis Esteban Velasco . . . . .	59
	<b>Bibliography</b>	<b>63</b>
<b>A.</b>	<b>Offline analysis details</b>	<b>67</b>
A.1.	Most relevant Clang-Tidy diagnostics . . . . .	67
A.2.	Cppcheck error list . . . . .	69
A.3.	UBSan error list . . . . .	74
A.4.	Table of code analysis . . . . .	76
<b>B.</b>	<b>Instructions for Docker setup</b>	<b>79</b>

# List of figures

1.1.	Gantt diagram portraying time management . . . . .	4
2.1.	Process of a classic online judge . . . . .	8
2.2.	DOMjudge logo . . . . .	8
2.3.	Report of a use after free error detected by Clang static analyzer through <code>scan-build</code> . . . . .	10
3.1.	Distribution of results in TAIS submissions (not including exam) . . .	14
3.2.	Verdict in test cases and submissions with ASan errors . . . . .	19
3.3.	Verdict in test cases with UBSan errors . . . . .	20
4.1.	Submission process sequence diagram. It contains the order in which the calls to the different scripts are made. . . . .	25
4.2.	Working directory structure of the judgehost . . . . .	27
4.3.	Example test cases for a problem . . . . .	29
4.4.	Option enabled from DOMjudge settings . . . . .	32
4.5.	DockerHub repositories . . . . .	38
4.6.	Containers view from Docker Desktop . . . . .	39
A.1.	Verdicts for <i>signed integer overflow</i> . . . . .	74
A.2.	Graph of verdicts for error: <i>execution reached the end of a value- returning function without returning a value</i> . . . . .	75
A.3.	Graph of verdicts for error: <i>pointer index expression with base</i> . . . . .	75



# List of tables

- 3.1. Number of diagnostics in compiled submissions . . . . . 16
- A.1. . . . . 67
- A.2. Static analyzers error’s explanations and communication . . . . . 78





# Introduction

In this section, we present the motivation of this thesis, as well as the proposed objectives, the work plan, a brief overview of this thesis' organization, and the repositories that include the software developed.

Online judges are widely used platforms to test the functionality of code for a specified problem. They are commonly used in competitive programming contests, but their application is also present in many educational institutions. Code is submitted to the online judge, which will be in charge of compiling the code and test it as an approximation to check its correctness; these tests usually involve inputting to the submitted code a predefined input and comparing the output against the expected one. The code can be written in any programming languages the online judge supports. Additional constraints, such as time and memory limits, can be imposed. Users will then receive the judge's verdict, *accepted* or *wrong answer* (the possible verdicts of DOMjudge appear in Section 2.1.1).

These are not the only two possible returned verdicts. A submission can be evaluated as inappropriate for a problem if it does not compile or execute correctly. Ensuring that a piece of code satisfies these conditions falls on the users themselves and, in many cases, it can be a non-trivial challenge (specially for those just starting to learn to code). One can attempt to read their code in an attempt to find errors on it. Verify where variables are defined and used, their types, the possible values that they can have during code execution, or what effect they can have on an action. This is, in essence, static analysis of code, and there exist many tools that perform this behavior on a much broader scale.

Static analyzers are great tools to find issues in code as it is written, as they do not require to execute the code. They perform a deeper analysis on the code than the one a person could reasonably develop, involving the examination of code structures, syntax, and semantics to detect issues such as inadequately used variables, bug prone code, and violations of coding conventions. They are commonly implemented in code development workflows to detect errors in the code early in the process. In addition to static analyzers, these type of workflows can also use code instrumentation, such as sanitizers, to complement the capabilities of static code analyzers. They are runtime tools (in contrast to static analyzers) and they can identify additional problems in the code, such as memory leaks and undefined

behavior.

Automatically analyzing a program and explaining its errors is an inherently difficult task, as semantic analysis of the code will generally be undecidable. This is proved by Rice's theorem, that states that all non-trivial semantic properties of programs are undecidable; non-trivial properties are ones that are neither true for every program nor false for every program. [28] So, it is impossible to assert that a certain program has a property just by looking at how it behaves (its semantics), unless that property is trivial. Moreover, an example of it can be seen in the halting problem, where it is demonstrated that it is impossible to write a program that, receiving as input another program, can decide whether it will stop or not.

However, various formal and informal techniques can provide partial answers to some of these problems. These methods have been restricted to the academy or the analysis of critical programs in the industry, but they have recently become more popular with the appearance of tools aimed at the standard programmer and the inclusion of analyzers in the most widely used compilers for languages as popular as C++. For instance, the latest versions of the GNU compiler and Clang include static analyzers to detect errors in the code without executing it, allow programs to be instrumented with predefined checks at runtime, and provide resources for advanced users to easily define their own checks.

These positive outcomes could be a welcome addition to the repertoire of possibilities that an online judge offers.

## 1.1. Motivation

In computer science, it is often the case that teaching institutions use practical laboratory lectures as complement to their theoretical ones, specially early in the degree. It is a popular and proven fact that one learns the most by doing, hence there exist many incentives to include online judges in laboratory classes as a main tool to apply this principle. Online judges can aid in structuring an active learning path by which students can apply their acquired knowledge during theoretical lessons to write code that tests their problem-solving skills.

The main disadvantage that this kind of tools bring to a studying environment are the often lacking feedback capabilities that they provide. Instructors are, in many cases, in charge of providing assistance to those that are stuck with a doubt or face a never seen before (often technical) compilation or execution error. Generally, it is unrealistic for instructors to be able to attend and satisfy every single student's needs, as it is to assume that multiple of them can be present simultaneously. There is even the case of no such instructor present or available at a given time, and having a tool as online judges, that are able to provide this sort of aid, despite not as specialized, can offer great value.

We, the team behind this thesis, have familiarity with these situations. Understanding the frustration that comes with not grasping the reason for an unaccepted submission or compilation error has determined to be essential in understanding the appreciation for this kind of interaction. There have been multiple reasons to require to perform a manual analysis of the code in order to identify the errors and

inconsistencies contained in it, essentially performing static analysis; the application of a tool that fulfills this purpose and provides feedback based on it is, in our belief, a valuable addition. A static analyzer is able to find errors in the code without executing it by interpreting the code following symbolic execution. These means that, as the values for the variables are unknown outside execution time, the analyzer must use a symbol to represent these values, and follow the possible paths of execution that the code can take. Each path contains a possible set of values that the variable could be assigned to, and if an error is found in that path, it is reported back to the user.

Static analyzers can be an additional step included during compilation or running of code inside an online judge, and its output can be used to generate a feedback message for students accordingly. They would serve well within them, specially when accompanied by additional tools such as code instrumentation. The difference between both tools is that the code needs to be executed for tools performing instrumentation to detect errors inside it.

Granted, instrumentation will, in contrast to static analyzers, consume more run time, but being able to customize the possible executions of code inside the judges themselves results in a very attractive proposition.

To add to the matter, code analysis is a field that extends to the early days of the appearance of programming languages, which has proven to contain lengthy research and many contributors to its development. Very advanced tools are present with outstanding performance and work behind, that could be implemented for the matter being treated.

By combining static analysis and linting with runtime instrumentation, achieving a more comprehensive approach to learning is possible. The challenge resides in applying the right balance to the feedback, and not give too much away as means of granting assistance, which would produce the opposite result that what is trying to be achieved.

## 1.2. Goals

Our main goal is to provide more meaningful feedback to student and instructors using online judges by leveraging on static analysis and instrumentation. At the end, we want to have them integrated into DOMjudge, but to achieve this goal we first need to understand the techniques and tools and find the most appropriate way to use them. We consider the following specific goals:

- Understand static analysis and instrumentation as provided by several mainstream tools.
- Study the behavior of a selected set of tools (Clang-Tidy, Cppcheck, ASan, and UBSan) on students submissions and decide which are most useful for meaningful feedback.
- Extend DOMjudge to allow the execution of code analyzers and instrumented submissions and presents their findings as feedback to students and instructors.

### 1.3. Work Plan

This project has been developed in two main phases: study phase and implementation phase. The time management for these phases can be seen in figure 1.1.

During the initial phase, we studied the necessary concepts in order to understand the basics of how static analyzers and instrumentation work and are used. Following this, a review on the most popular tools in that domain was carried out; this process was fundamental in order to determine which would be included later in the implementation. Once tools were chosen, we analyzed their performance on a set of real student submissions provided by our tutor. This data was part of real student submissions for exams and class problems, which was beneficial because it meant that we could perform an analysis on actual real submissions in order to obtain representative data.

Following this, the first step towards being able to implement these tools was understanding how DOMjudge receives a submission and judges it. This is a necessary step to identify the relevant code that should be modified in the upcoming steps. Secondly, we prepared and set up a Docker environment to deploy a DOMjudge instance in our machines for testing and developing purposes. Once the required changes were included in our extended DOMjudge, we put everything done so far together to specify the right checks that should be included for the static analyzers and instrumentation.

With this, it remained to verify the functionality of the modified judge with realistic submissions to check the correctness of the implementation and document everything.

The following Gantt diagram portrays the time management during this whole process:

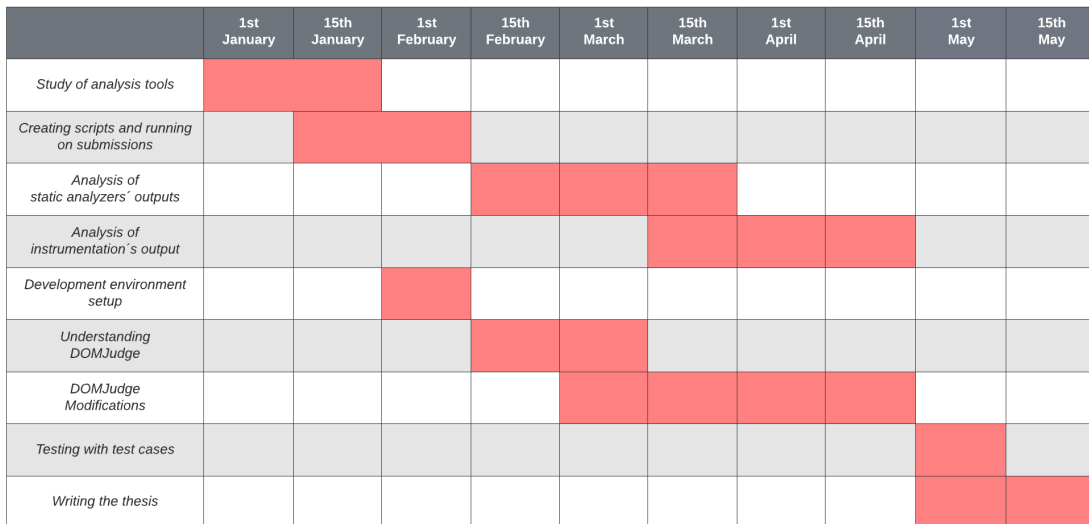


Figure 1.1: Gantt diagram portraying time management

## 1.4. Organization of this thesis

The general structure of the thesis consists of the following sections:

- Chapter 1 (Introduction): This part sets out the reasons for carrying out the work, the objectives to be achieved, the organization of the thesis as well as linking the repository of the code used in order to carry out this thesis.
- Chapter 2 (State of the art): In the state of the art we focus on explaining the platform we are going to modify and the tools that we are going to use for the realization of this thesis.
- Chapter 3 (Offline analysis on real student submissions): This chapter we analyze the output of the static analyzers and sanitizers on real submissions. We also show the procedure to obtain and analyze the data, the results of that analysis and the conclusions we have arrived to.
- Chapter 4 (Integration of static and runtime analysis in DOMjudge): this chapter focuses on the modifications made to the DOMjudge platform to integrate the tools necessary for creating the feedback.
- Chapter 5 (Real use case scenario): In this chapter we show two tests carried out in order to verify the judge's functionality. Firstly, a prepared submission that serves as proof of concept and, secondly, a made up submission that attempts to be as close to reality as possible. We will show the feedback provided by the judge.
- Chapter 6 (Conclusions and Future Work): This chapter sets out the discussion of the thesis where we discuss the main problems we encountered and how we solved them, it also shows the conclusions we have arrived to by developing this thesis as well as the future work that could be carried out.
- Appendix A: table containing an in depth list regarding occurrences of Clang-Tidy diagnostics in submissions, the complete list of relevant checks for the tools of Cppcheck and UBSan as well as a table comparing all the code analysis checks to not overlap them.

## 1.5. Source code and data repositories

We have developed and maintained a series of repositories during the development of this project that collect our efforts to complete the different milestones.

The GitHub repository contains the most relevant scripts and code that we have developed for this thesis: scripts for analyzing the information provided by the tools used in the student submissions, scripts for parsing the output of the analyzers, code for creating graphs to understand better the results of the analyzers, the code of the test case and real use case scenarios, the parsers used in DOMjudge to show the feedback, and the composition of the Docker:

`https://github.com/Jantri-3/analysis-of-DOMjudge-submissions`

This repository is the result of a common effort of the whole team.

The modifications applied to the source code of DOMjudge to integrate our code analysis features is available in a fork of the official repository at:

`https://github.com/robu02/DOMjudge`

The changes made to the source code will be explained in Chapter 4. Moreover, Docker Hub repositories have been used to maintain Docker images of the modified DOMjudge. These images have been created by us from the modified DOMjudge source code. The DOMServer image is at

`https://hub.docker.com/r/robuso02/domserver`

and the judgehost image at

`https://hub.docker.com/r/robuso02/judgehost`

The creation of these images will be explained in Section 4.4.

# Chapter 2

## State of the art

This chapter will cover the main aspects of the thesis to understand online judges platforms and the tools we are going to implement on the DOMjudge in order to give feedback to the student.

### 2.1. Online judges

The online judges were created to facilitate code corrections for instructors at universities, we will start by defining what is an online judge. Online judges are platforms (usually with web interfaces) used for code assessment playing a pivotal role in education, competitive programming, and career training. [21]

Online judges take examples of inputs and their corresponding expected output prepared by the instructor, then introduce that same input to the code uploaded and compare the output with the expected one to check if it matches, if it does, the judge qualifies the submission as correct Figure 2.1.

Some judges used in education institutions in Spain are DOMjudge and Jutge.org [27]. Jutge.org is developed by the Polytechnic University of Catalonia in order to provide a platform for learning and practicing programming (tailored specifically for their computer science students).

As mentioned before, online judges are also used in programming contests. Some notable examples are: the International Collegiate Programming Contest (ICPC) [12], an algorithmic programming contest for college students where DOMjudge is used, and the International Olympiad in Informatics (IOI) [25] contests for secondary and high school students where the judge used is CMS [5], as well as their regional qualifiers.

Some other famous online judges are Project Euler [16], Codechef [6] and DMOJ [2].

In this project, we will discuss DOMjudge Figure 2.2, the official judge of the ICPC, as it is the one we use at our University.

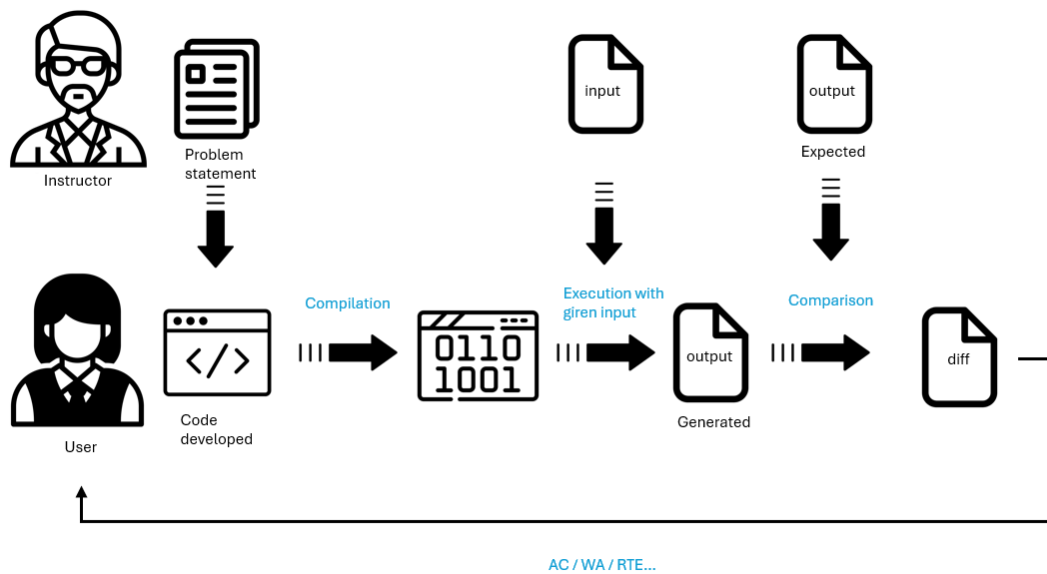


Figure 2.1: Process of a classic online judge



Figure 2.2: DOMjudge logo

### 2.1.1. DOMjudge

As stated on their website, “DOMjudge is a system for running programming contests, like the ICPC regional and world finals programming contests.” [7]. However in the specific case of universities, it is used for giving students assignments and to correct them by an automatic process. These assignments consist of coding problems that read input according to the problem input specification (given by the problem setter) and try to make a code that writes the correct, corresponding output.

To return a verdict, the judge performs several steps on a submission. Firstly, it will compile the code (if not an interpreted language), since the users provide their source code. Compilation is performed based on the programming language of the submission, which is specified by the user at the moment of submission. The resulting executable file is then executed, receiving as input the test case specified for the problem attempting to be solved. The output produced during the execution phase is saved to be compared with the expected output of the problem. The output will be parsed to verify its similarity with the solution.

The previous steps can be customized to the needs of the problem. DOMjudge



offers the possibility of using different compare and run files. If they are not specified, defaults will be used. These special run and compare files allow for specialized and specific checks of requirements or steps for a judgement of a specific problem. To be used, they have to be selected in the `special_run` and `special_compare` fields of the problem.

Depending on the results of the execution and the expected output, a verdict is obtained for each test case outputs a result, these results are then aggregated into a main verdict. The verdict given can be:

- WA (Wrong Answer): the code produced an output different from the expected.
- CE (Compilation Error): the code has compilation errors.
- RTE (RunTime Exception): during the execution of the code an error has occurred.
- TLE (Time Limit Exception): the time limit for code execution has been exceeded.
- AC (ACcepted): the code has finished within the time given and has produced the expected output.

Although these verdicts are not frequent, they may sometimes happen:

- NO (No Output): the code has produced no output.
- OLE (Output Limit Exception): the code produced too much output.

When there's more than one test case then the results are given based on the first highest priority result. This priority is decided by the administrator of the judge, in our case the priorities are:

AC: 1  $\ll$  NO: 10  $\ll$  WA: 30  $\ll$  TLE, RTE, OLE: 99 (maximum priority)

Then, if WA and TLE are the results in the test cases of a problem the submission result would be TLE as it has higher priority.

Some of the features that DOMjudge provides are: Web interface; Modular system for plugging in languages/compilers and validators; Jury information (such as number of submissions, judging and diffs) and options (rejudge, clarifications, resubmit). However, there is no actual tool provided for giving detailed feedback, they just give the student the result as explained before.

The main developers of this platform are Jaap Eldering, Nicky Gerritsen, Keith Johnson, Thijs Kinkhorst, Mart Pluijmaekers, Michael Vasseur and Tobias Werth.



Clang-Tidy, is a linter tool, which checks for style violations, coding standards, possible performance upgrades and logic errors. Checks are mostly implemented via AST matching, but some of them just work using text matching. The different checks it performs are divided into families, each one of them has a distinct prefix. For example, checks with the clang-analyzer prefix belong to the same family, the one containing the Clang static analyzer checks and Checks starting with clang-diagnostic are warnings that the compiler generates. Other families can be related to a specific project's style guidelines, to bug prone code, to inefficiencies or to modernization of the code. See a full list of checks [34].

### 2.2.2. Cppcheck

Cppcheck is the other static analysis tool we are using, it was developed in 2017 for C and C++ code. It is used by developers to correct their code as it identifies potential bugs, and security vulnerabilities in their code by analyzing the source code without executing it. There is an open source version (Cppcheck) and a more refined paid version (Cppcheck Premium). Cppcheck transforms the source code into an internal representation that captures the syntactic structures of the code. This representation is similar to an AST<sup>1</sup> and allows Cppcheck to understand the hierarchy and relationships within the code as well as to track variables and their state throughout the code, detect conditions that may cause errors, such as buffer overflows, invalid memory accesses, and other problems. Cppcheck also does a constant variable propagation, which implies that if for example, in the code the variable  $x$  always have the same value (5) instead of writing  $x$  in the "AST" (is not completely an AST but a similar structure as said before) it writes directly the 5.

In this work we have used the open source tool, which has a large number of checks that can be run. To see a full list check the following reference [23].

## 2.3. Instrumentation

Instrumentation is the other way to analyze a program. Unlike static analysis, which does not execute the code to analyze it, this technique works by running a perhaps modified program and then analyzing what it does during or after execution. It works by modifying the program to add code meant to measure dynamic behavior and log information about it. It can be used to detect errors in the memory management, undefined behavior, performance bottlenecks, etc. It is limited in the sense that it can only detect errors in the code that is executed. It also needs test cases in order to run. On the other hand it is more efficient and easier to implement.

As we are doing analysis on C++ code we looked into tools dedicated to it such as AddressSanitizer, UndefinedBehaviorSanitizer, Valgrind [4], an open-source instrumentation framework with tools for memory and threading error detection, Purify [31], a proprietary memory error detector, and Quantify [32], which analyzes

---

<sup>1</sup>An Abstract Syntax Tree (AST) is a hierarchical and structured representation of the source code that captures the syntactic structure of the code in a way that facilitates its analysis. Each node in the tree represents a language construct, such as operators, statements, expressions, etc.

performance (also proprietary), but there are plenty other instrumentation tools for different programming languages. Some languages and libraries also include runtime checks, like array bound checking, in their programs, either on demand or by default.

We decided to use `AddressSanitizer` and `UndefinedBehaviorSanitizer` because they are faster than `Valgrind`, which will ensure the feedback reaches the students faster, open-source and easy to use as they come with the Clang and GCC compilers.

### 2.3.1. `AddressSanitizer`

`AddressSanitizer` or `ASan` [33] is a memory error detector. It consists of a compiler instrumentation module and a run-time library. It comes with the Clang, MSVC, and GCC compilers, and it is open-source. The sanitizer works by replacing the `malloc` and `free` functions, it marks some areas of the memory so when an access is made to that area it is reported as an error. The message produced contains the type of error it is, for example stack overflow or segmentation fault, the stack trace, and, when relevant, if the error is caused by a read or write access or some hints about it. It typically introduces a x2 slowdown. It only reports the first error it encounters, then the program is stopped, as further errors could be caused by the first one.

### 2.3.2. `UndefinedBehaviorSanitizer`

`UndefinedBehaviorSanitizer` or `UBSan` [35] is a open-source code sanitizer implemented on Clang and GCC. Undefined behavior is behavior unpredictable under the C++ standard, as the language specification to which the code adheres does not prescribe a result. It is typically the result of the broadening of requirements so that compilers are able to make optimizations.

The compiler inserts instrumentation that performs certain kinds of checks before operations that may cause undefined behavior like null pointers or integer overflows.

UBSan detects array subscript out of bound where the bounds can be statically determined, bit-wise shifts that are out of bounds (for their data type), distinguishing miss-aligned or null pointers, signed integers overflows, and conversion between floating-point type which would overflow their destined types.

A full list of UBSan available checks can be found here.[35]

## Offline analysis on real student submissions

In this chapter we will analyze the output of the static analyzers and sanitizers on real submissions. We will show the procedure to obtain and analyze the data, the results of that analysis and the conclusions we have arrived to.

### 3.1. Context of the course

*Software engineering algorithmic techniques* (TAIS) is a course taught in the Universidad Complutense de Madrid to third year software engineering students. In this course, they provide an online judge for students to practice on during the whole semester. In the academic year 2023/24 76 students were enrolled to this course. The dataset comprises submissions sent throughout the course and in the final exam. Throughout the year they sent their solutions to 81 different problems. The exam consists of 2 problems. We can see a quick overview of the judge's verdicts to the dataset submissions on Figure 3.1.

In total there were 4119 submissions, but only 3663 successfully compile.

Teachers recommend being sure that the code compiles and gives the correct output when tested with the input specified in the problem explanation. So it is expected that we observe the more common results are AC and WA, as other results are related to harsher errors, which are often detected by local testing, or are just less frequent overall.

### 3.2. Data extraction and manipulation

Although for the different tools we have followed different procedures, all of them had some parts in common. First, we studied all the available checks that the tools provided. Then, in order not to disclose the identity of students, our tutor ran the tools on the submissions and provided us with the results.

At that point, we decided to focus on the checks we would analyze based on the number of occurrences on the submissions and its relevancy regarding student. In

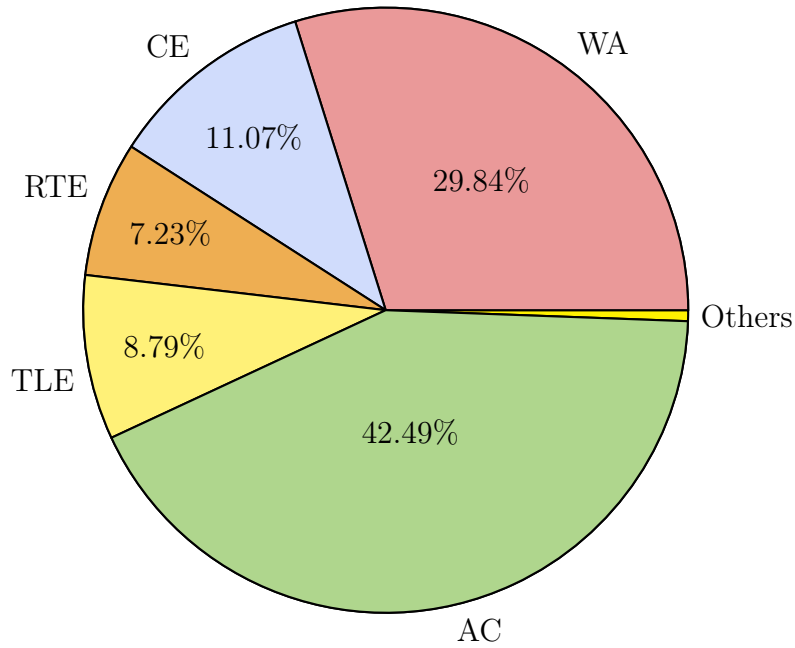


Figure 3.1: Distribution of results in TAIS submissions (not including exam)

the case of the sanitizers, we studied all the possible errors as they are few.

Then, to analyze which of those checks were actually relevant, we made different scripts (one for each tool) in which we tried to discern their relevance to the feedback. This was done by extracting the warnings, counting them and making statistics to check the relevance.

The scripts' functionality is to join all results of the tools and then check the distribution of judge results by the error's ID. With another script we also counted each error occurrences and transitions to AC (submissions in which the error disappeared when the result changed to AC), which can help identify errors that cause the submission to be rejected.

The main difference between the scripts is the input format of each tool, as Clang-Tidy used a YAML file, Cppcheck a XML file, and ASan and UBSan just text, which is then parsed by detecting characters that are always in the same position and using regex. All of those outputs needed to be parsed differently by the scripts to then print the relevant information we mentioned earlier.

With the data extracted, we wrote some Jupyter notebooks to express in a visual way the data received. The scripts and the notebooks are included in the repository, in the `analyzers` and `graphs` directories respectively <sup>1</sup>.

### 3.3. Code analysis results

In this section, we show the results of the static analysis divided by tools.<sup>2</sup>

<sup>1</sup><https://github.com/Jantri-3/analysis-of-DOMjudge-submissions>

<sup>2</sup>After exhaustive investigation we saw that GCC static analyzer checks were not interesting errors, so we decided to leave them apart.

### 3.3.1. Clang-Tidy

From the static analysis we extracted the number of occurrences of each diagnostic, for the more relevant checks you can see Table 3.1, or the full table in Appendix A. We also extracted data about the distribution of results from submissions containing each diagnostic and the number of transitions to AC, which is available in the repository<sup>3</sup>, in the `analyzer_output` directory. Before diving into the results, we need to understand each of the relevant diagnostics.

After removing the checks that can't be of help to students, leaving aside checks concerning clean and modern code for the future, as we will not be able to extract too much information regarding its effects on the output and then removing those that appear only a couple of times we end up with checks of the following families: bugprone, which often result in errors or unexpected behavior that is difficult to detect by the programmer, performance, which refer to working code that might take a lot of time to run, maybe resulting in a TLE and one diagnostic belonging to the misc category. We also added all the Clang static analyzer checks that appeared. See the errors selected explained below.

- **performance-avoidendl**: `std::endl` performs 2 operations, writing a newline character to the output stream and flushing it. Flushing of the stream can be inefficient when making many I/O operations.
- **bugprone-narrowing-conversions**: Checks for silent narrowing conversions like `int i = 0; i += 0.1;`
- **performance-for-range-copy**: C++11 range-based loops where the loop variable is copied in each iteration and obtaining it by reference would work.
- **performance-unnecessary-value-param**: A parameter of an expensively copiable type is passed by value when only constant methods are used. Hence, it can be passed by constant reference.
- **clang-analyzer-deadcode.DeadStores**: Values stored to variables that are never used afterwards.
- **bugprone-branch-clone**: Branches in conditional statements that are identical.
- **bugprone-suspicious-semicolon**: Instances of stray semicolons that unexpectedly alter the meaning of the code. For example one at the end of an if statement. For example, `if(x == 5); {x++;}` Will always increment x value.
- **clang-analyzer-cplusplus.NewDelete**: Double free and use after free errors.
- **misc-redundant-expression**: Redundant expressions which are typically errors due to copy-paste. For example, `if(x == x)` is always true, but the writer probably meant for another comparison to be made.

---

<sup>3</sup><https://github.com/Jantri-3/analysis-of-DOMjudge-submissions>

Table 3.1: Number of diagnostics in compiled submissions

Name	Submission	Team	Problem
performance-avoid-endl	564	263	48
bugprone-narrowing-conversions	481	241	38
performance-for-range-copy	117	64	11
performance-unnecessary-value-param	78	56	21
clang-analyzer-deadcode.DeadStores	57	32	15
bugprone-branch-clone	56	25	12
bugprone-suspicious-semicolon	38	16	11
bugprone-integer-division	32	28	1
clang-analyzer-cplusplus.NewDelete	27	13	2
misc-redundant-expression	14	3	1
clang-analyzer-optin.cplusplus.UninitializedObject	1	1	1

- **clang-analyzer-optin.cplusplus.UninitializedObject**: Uninitialized fields in objects created after a constructor call.

In Table 3.1 we show the number of occurrences of each diagnostic within all submissions, shown in the second column. But, in order to avoid any kinds of bias related to diagnostics appearing multiple times within a team or a problem we added two other columns. The third column counts the number of team’s submissions for a problem which contained the diagnostic, and, finally, the fourth column contains the number of problems which have submissions containing the diagnostic. See the full version of Table 3.1 in Table A.1.

From the Clang-Tidy statistics we can reach the following conclusions for each of the relevant diagnostics:

- **performance-avoid-endl**: It has 18 transitions from TLE to AC and even if the 43% of AC submissions is similar to the 48% global percentage of AC, as it has the most transitions to AC, and it is a nice opportunity to teach students more about IO operations, it would be a nice one to communicate to students using the judge.
- **bugprone-narrowing-conversions**: It also has a percentage of AC very similar to global one due to not always being a critical error, but it is probably always unintentional and it also appears in many problems.
- **performance-for-range-copy and performance-unnecessary-value-param**: As it was expected the performance do not appear to have relevancy when looking at AC rates or transitions, but as they teach an important lesson and appear many times, we have decided it is beneficial to communicate it to the students



- **clang-analyzer-deadcode.DeadStores**: It has a high acceptance rate (38%) because it can be due to a leftover value that is not used in the end, but, it has 3 transitions to AC showing it could be helpful to fix errors in some submissions.
- **bugprone-branch-clone**: 37% of AC submissions is not significant enough, but as many times it is an error due to copy pasting and then forgetting to change something, it could be communicated with lower priority.
- **bugprone-integer-division**: It has a very low acceptance rate, but it only appears in one problem, could be communicated only in some problems, it is also hard to spot if you do not know about it.
- **bugprone-suspicious-semicolon**: It is not always a critical error but as it was observed in some submissions it is very tricky to spot. Some students abandon the problem before reaching the solution and others reach the solution with the diagnostic still present, for example a student used a ternary operator within an if statement with a semicolon afterwards, this is why this diagnostic has a low number of transitions to AC.
- **clang-analyzer-cplusplus.NewDelete**: It has 0 transitions to AC and a high acceptance rate (48%). This is due to some false positives.
- **misc-redundant-expression**: We decided to include this check even though it appears a very small amount of times, due to the fact that it is very tricky to find sometimes, it also has 0 transitions to AC, but that is due to teams abandoning the problem before reaching an AC submission, one team went as far as submitting 11 solutions without reaching an AC. When analyzing those submissions code we noticed that it was a logic error that was easily solvable.
- **clang-analyzer-optin.cplusplus.UninitializedObject**: It appears only once in a submission with RTE.

### 3.3.2. Cppcheck

Cppcheck checks that appeared on the dataset are divided into 5 major categories:

- **information** : 3 types of checks appeared, just 1 is important
- **style** : 24 types appeared, 13 of them relevant ones
- **warning** : 7 types appeared, 6 are important
- **performance** : 4 types appeared, all of them are didactic
- **error** : 8 types appeared, just 3 are actually relevant

Cppcheck is a extensive static analyzer as explained in the state of the art. This is why, we can't find any similar checks between each other. However, as they are quite specific, we can actually join various checks into just one. This can be

seen on the table of appendix C, where we make a detailed explanation of each Cppcheck and Clang.tidy check and determine that 4 different Cppcheck's checks can be covered by just one Clang-Tidy check. The checks were "**redundantCondition**", "**multiCondition**", "**duplicateExpression**" and "**constStatement**", all of them could be simplified by letting the Clang-Tidy check **misc-redundant-expression** localize and explain to the student the flawed coding practice they had. This can be seen on the table in the Appendix A.4.

After an exhaustive study of the checks, we have reduced the number of checks raised from 46 different relevant checks to just 27 relevant checks. All relevant checks cover completely different areas or bad practices of the code (this is a result of the said characteristics of Cppcheck). The checks are very diverse and could not be reduced anymore as they did not overlap at all. However, there is actually 3 checks that cover different "areas" of the code but could be easily joined into one, The **ShadowVariable**, **ShadowFunction** and **ShadowArgument**. As the names suggest, they are all related to collisions on named variables/functions or arguments used on an inner function which overlaps the meaning. This is a poor programming practice and must be communicated to the student.

The following section is included in the appendix rather than the main text, not due to its lack of relevance, but to avoid overwhelming the reader. Although this information is quite interesting and influential, the format is repetitive as it details all aspects of the errors identified by Cppcheck. Nonetheless, we strongly recommend dedicating time to the Appendix A.2.

## 3.4. Instrumentation results

In this section we show the instrumentation's results of each of the sanitizers

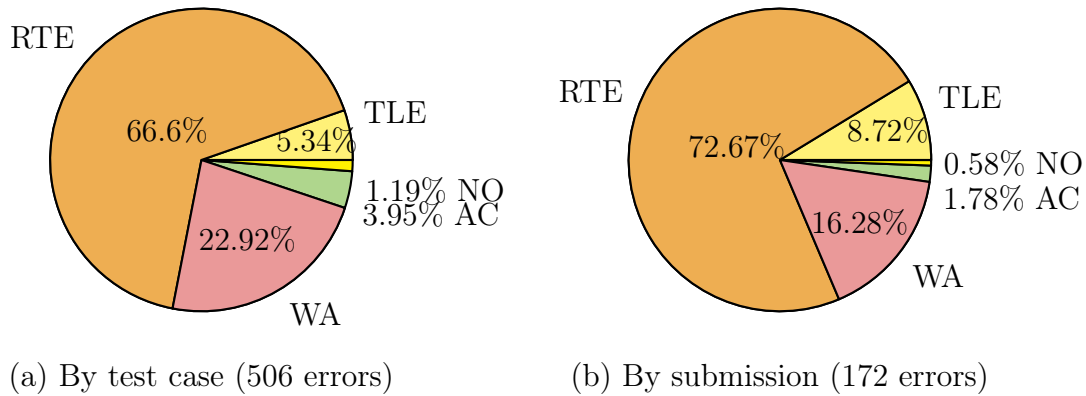
### 3.4.1. Memory errors by ASan

Out of 3663 submissions, 172 have ASan errors. On average, there were 2.94 errors per submission, for a total of 506 errors. You also have to account that each problem has a different number of test cases and ASan only generates output when faulty code is reached. From the ASan outputs we extracted data about the number of each error's occurrences, about the distribution of results on each test case with an error, and on each submission with errors, and about the transitions from a not accepted test case to an accepted one when the ASan errors are fixed. They can be found in the repository, again in the **analyzer\_outputs** directory.

We can observe that most of the submission with ASan errors were rejected by the online judge 3.2, the errors that appear the most are heap buffer overflows, segmentation faults and stack overflows, and they are not committed by a few teams, nor in a few problems, meaning they are common errors. Runtime exception is the most common result for all errors, except for memory leaks, this is expected as they do not usually cause the program to stop, unless it runs out of memory.

Surprisingly, there are submissions with fatal ASan errors that are accepted by the judge. Out of the 20 test cases with errors that were accepted: 2 were memory

Figure 3.2: Verdict in test cases and submissions with ASan errors



leaks and 18 were heap overflows. Although memory leaks should be avoided, they are not errors which should be fatal to the execution of the code, as it should continue execution if there is enough memory. Heap overflows on the other side are more dangerous, they are due to out of bounds accesses, they only stop execution if the position besides the vector is accessible by the program. Even though that position in memory is not supposed to be accessed, the program can luckily read the right value or store it in variable that is not used, hence the solution can still be accepted.

### 3.4.2. Undefined behavior by UBSan

There are 408 submissions out of a total of 4119 that generate one or more `ubsan.log` files containing the errors. Generating a total of 930 `ubsan.log`.

On average 2.28 files are generated per submission, again there are problems with more cases than others. As we have repeated errors, we need to eliminate any redundancies found. We can find several errors on an assignment repeated on all test cases of said assignment. Within the same case, we can have several errors, so we needed to make a Python script to clean the data.

After cleaning the data, we need to make even more modifications to the data in order to associate errors by type as UBSan distinguishes errors by details that the students don't need (such as "null pointer with base `0xACDFFFFFFF...`"). We managed to identify 14 different types of errors via regular expressions.

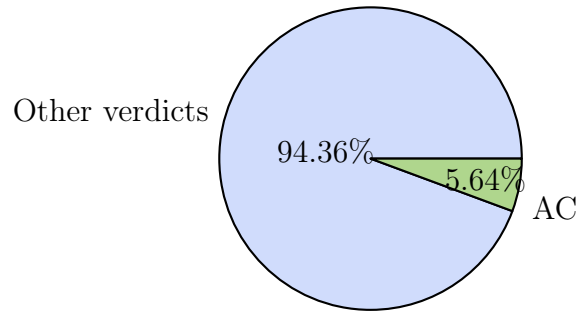
The majority of UBSan errors detected result in not accepted qualification by the judge with only 23 out of 408 submissions being accepted. Figure 3.3

UBSan errors detected should be fatal for the execution in any case (meaning that they will probably interrupt the execution). However, the programmer could be "lucky", as the memory space not intended by the programmer for reading could have a not null value or contain something which made sense in the code. This is the main reason that all UBSan errors raised should be notified to the programmer.

It is clearly observed when checking the errors raised that most errors are related to pointers. Although this is expected, it is interesting.

Every time a "load of null pointer", "error reference binding to misaligned ad-

Figure 3.3: Verdict in test cases with UBSan errors



dress”, “load of misaligned address”, “store to null pointer”, “store to misaligned address”, “member call on null pointer” and “null pointer passed as argument, which is declared to never be null” appears on the `ubsan.log`’s generated, the result of the judge on that test case is always **RTE**. These results make sense as the undefined behavior errors are usually fatal at runtime as said before (at least with the previously mentioned, this is not the case for example with integer overflows). An example of interest is “store to null pointer” which remind us of some errors ASan also detects as “segmentation fault” which is a memory address error. These errors can appear together when running both tools on the same submission.

To see the rest of the list of UBSan checks please refer to the Appendix A.3.

### 3.5. Conclusions

After an exhaustive analysis of the static analyzers and the sanitizers we conclude that they are of great importance in order to provide a detailed feedback to the students, with these different tools, Clang-Tidy, Cppcheck, ASan and UBSan, we cover some common mistakes students make with a list of 53 checks.

We have observed that a submission having errors detected in the static analysis usually means that the submission was rejected by the judge. Submissions with ASan errors are accepted only 1.74% of the times (3/172), many times due to a runtime exception, as observed in figure 3.2. Submissions with UBSan errors have an even lower percentage of acceptance 0.56% (23/408). Even if they only appear in a low amount of submissions. On the contrary, linters diagnostics appear on a higher number of submissions but are less prone to not being accepted, for example many Clang-Tidy diagnostics have a percentage of acceptance similar to the total submission acceptance rate, which is 42.5%. This means both kind of tools will complement themselves, while instrumentation will detect harsher errors that are easier to detect at runtime, static analysis will detect errors that are easier to detect when analyzing the source code or inefficiencies that when solved will make for a better solution, even if they do not affect the judge’s result.

# Chapter 4

## Integration of static and runtime analysis in DOMjudge

This chapter will focus on the actual changes performed to the DOMjudge source code in order to achieve our goal. For this purpose, one has to study and understand how the judge works at a more specialized level in order to see what changes must be performed.

There was an abundance of debugging of errors, with their corresponding investigation, to find workarounds to issues in this section. Here, DOMjudge developers' insights and advice served as great paths to follow to get through them, and they will be discussed and mentioned across the corresponding segment of the chapter. The first and most important step would be to identify the appropriate moment to run the static analyzers and instrumentation during the evaluation process, as well as the location in the file system where the files must be stored. This must not obstruct –or do in the least amount possible– the judge's tasks that must be carried out in order to evaluate a submission. Once an appropriate method is discovered, the next logical step would be to find the way to get the static and runtime analyses outputs to the user.

To carry this task out, Docker has been very helpful. The ease of deployment of Docker containers allowed for a very permissive testing suite that could be dismantled and set back up as many times as necessary. It also allowed for every team member to replicate the development workspace in their machines.

In section 4.1, we will focus on dissecting how the DOMjudge works and its structure. We will how a submission is evaluated and the processes that it has to go through. Then, in section 4.2, we will explain the implementation decisions that were made once the submission process was understood, and the necessary changes that had to be applied in the source code to make them happen. During section 4.3, we will then explain how a message for a submission is built based on the outputs generated by the static analyzers and instrumentation that have been implemented and how it is presented to the student. Finally, in section 4.4, we will expose how Docker was used in order to achieve all of this, navigating the process of generating a new image and what was required in order to set up. Also, we will explain the creation of a `Docker-compose` file that orchestrates the process of initializing all

the containers in an automated manner in order to avoid having to initialize them manually.

## 4.1. More on the DOMjudge architecture

As the first milestone to achieve our goal, understanding how the judge works is crucial. Its components, requirements, which sections are relevant to us, and following the order of calls. The goal is set mainly on understanding how submissions are processed, as other areas of the judge are not relevant for this project. We are interested in knowing what happens from the moment a student presses the submit button, to the moment that the evaluation is returned. Where does it compile? How does it execute? How are the comparisons made between the expected and actual output? The answers to these questions would give us the clues to know where to start making the necessary modifications. Since we're dealing with static analyzers and sanitizers that have to be included during program compilation, it's specially relevant for us to know where and how this function is carried out.

The DOMjudge version used at the Computer Science School's laboratories at the time of conducting this project is a rather outdated version. DOMjudge is an open source project and, as such, it is constantly receiving updates thanks to the involvement of its main developers and the participation of volunteer contributors. The source code is published on GitHub,<sup>1</sup> and the project counts with a Slack workspace for any type of conversation related to it, a mailing list for development discussion—which has been used for this project—, and an inbox for bug reports or feature requests through GitHub or the mailing list itself.

It is worth noting that DOMjudge is primarily designed for its use in programming competitions that take place over a period of time that can range from 5 hours to a weekend. As such, its use in the classrooms is not the intended original purpose, and it is important to understand this in order to grasp the reasons for the absence of any learning functionality in its implementation. Our goal will be to modify the source code to achieve features that are not supported and may trigger incompatibilities with the rest of the implementation. We'll delve deeper into the various issues encountered throughout the project in the sections that follow, as well as some of DOMjudge's implementation details.

### 4.1.1. The structure

DOMjudge is composed of two main components, DOMserver, a database and at least one judgehost. They are independent components that communicate with each other.

DOMjudge is composed of two main components: DOMServer and at least one judgehost. They are independent components that communicate with each other. The different judgehosts are in charge of compiling, running, and checking a submission.<sup>2</sup> Aside from these, the actual installation also requires a database to store

---

<sup>1</sup><https://github.com/DOMjudge/DOMjudge> [9]

<sup>2</sup><https://www.DOMjudge.org/docs/manual/8.2/judging.html#flow-of-a-submission> [36]

all the information available inside the server. It communicates directly with the DOMserver component.

DOMserver is short for DOMjudge server, and it is the central entity that runs the DOMjudge web interface and API; which is based on the Contest Control System (CCS) Contest API specification. The DOMserver uses a MySQL or MariaDB database server for information storage. For the development of this project, we will be using a MariaDB database instance. The machine in which the DOMserver is hosted needs to be running Linux or Unix variant and a web server with PHP 8.1 or newer.

For the online judge to be functional, it requires one or more judgehosts which will perform the compilation and evaluation of submissions. For our testing purposes, a single judgehost will suffice, as the DOMjudge team recommends one judgehost per twenty teams.<sup>3</sup> Judgehosts will connect to the server and wait for a solution to be submitted by a team. Submissions are added to a queue from which the different judgehosts will retrieve them and then return a verdict after the required processing has been done on it. The judgedaemon is in charge of managing this inside the judgehosts [30].

#### 4.1.2. Submission environment

The judgedaemon compiles and executes submissions inside a chroot environment<sup>4</sup> for security reasons – security is a main concern for the team, as arbitrary code is being executed in the system. This environment must contain all the requirements that are needed at compile time to build and run the submissions. DOMjudge also uses Linux Control Groups (or cgroups) and namespaces for process isolation in the judgedaemon. Cgroups allow to allocate resources such as CPU time, memory, and network bandwidth to user-defined groups of tasks.<sup>5</sup> This aspect limited some of our initial options, since we considered the use of API requests from the scripts ran inside the execution environment in order to retrieve additional useful information, such as the number of attempts for a problem by a team. Due to the fact that the execution environment sees its Internet access limited, the option had to be scrapped.

#### 4.1.3. Submission process

When a student or team makes a submission, this submission is added to a queue of not yet evaluated submissions. When multiple judgehosts exist, they can all work in parallel to evaluate said queue, each performing their actions on individual submissions. Each submission will be picked by one of the active judgehosts and it

---

<sup>3</sup><https://www.DOMjudge.org/docs/manual/8.1/overview.html#requirements-and-contest-planning> [36]

<sup>4</sup>A chroot environment can modify the root directory for a process, restricting access to the rest of the file system. The command being run has no idea that anything outside its environment exists. <https://www.howtogeek.com/devops/what-is-chroot-on-linux-and-how-do-you-use-it/> [15]

<sup>5</sup>[https://access.redhat.com/documentation/es-es/red\\_hat\\_enterprise\\_linux/6/html/resource\\_management\\_guide/ch01](https://access.redhat.com/documentation/es-es/red_hat_enterprise_linux/6/html/resource_management_guide/ch01) [19]

will be evaluated by it. To do so, the code traverses the *compile*, *run*, and *compare* phases. Execution is performed in the previously mentioned restricted environment. Restrictions in the environment will be mainly in the form of time and memory usage limitations. The output produced during the execution phase is then compared with the expected output of the solution, which is the second step of this process, where all the compare process takes place. Once completed, processing on the judgehost terminates, and it returns the submission information to the DOMserver, in charge of displaying it to the user.

The entire process is orchestrated by the code contained in `judgedaemon.main.php`, running on every judgehost. It evaluates the submitted assignment and transfers the results back to the DOMServer. The tasks of compiling, running and comparing the code are delegated to a number of different scripts that generate the required files. First, `compile.sh` is executed to compile the submission. The respective executable path is passed to `testcase_run.sh`, whose task is to perform run and compare on the submission. Modifications performed to this second file have been key in order to achieve the goals of this project, as it plays a main role during the evaluation process.

Before continuing, we will clarify some terms that may induce confusion. The execution or run phase of the submission process is done by executing a script called `run`. To make it clear as to when we are referring to the run phase and when we are talking about the `run` script, we will always refer to the phase explicitly. Additionally, the compare phase is performed by executing an executable called `run`, that is obtained when compiling the source code `compare.cc`. To avoid confusion between both `run` files, we will refer to the `run` produced for the compare phase as `compare` from this point onwards, and the compare phase will always receive the explicit distinction.

The whole process that involves compiling (for our particular case, this will always take place, since we are using C++ code), running and comparing the results is executed for every test case specified for a problem. This is because of the multiple judgehosts can exist at the same time, and the different test cases can be distributed among them. Compilation will only take place once per judgehost, as the judgehost can identify if it already exists a *compile* output, as means to save time. There can exist one or more than one test cases per problem, and the `testcase_run.sh` script is run for every one of them.

Figure 4.1 visually represents the order of calls that are performed for the described process.

#### 4.1.4. More on files

It has been indicated which are the files in charge of performing the different actions, and in this section they will describe the scripts `compile.sh`, `testcase_run.sh`, `run`, and the executable `compare` further.

**compile.sh** This is an internal DOMjudge script responsible for compiling the student's assignment and is called from `judgehost.main.php`. Being internal means that it is not meant to be changed by the users. It accepts as input the absolute path



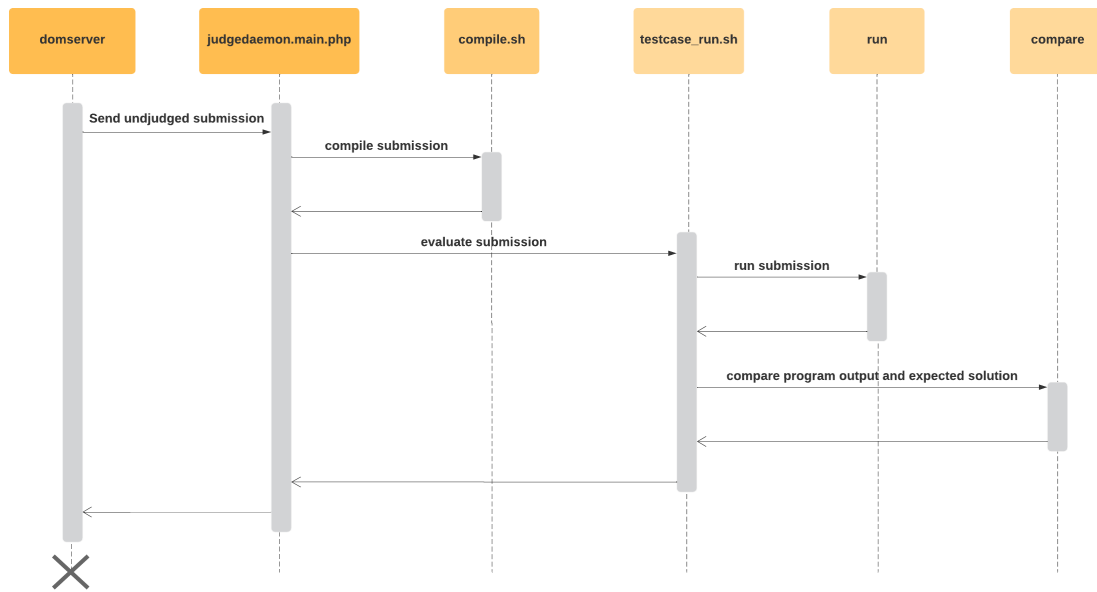


Figure 4.1: Submission process sequence diagram. It contains the order in which the calls to the different scripts are made.

to the compilation script for the language, the working directory for the evaluation of the current assignment, and the file or files submitted by the student. It returns 0 in case of success, “compiler-error” or “internal-error” if defined, or 1 otherwise. This is the `compile` wrapper-script called from `compile.sh` to compile C++ code by default:

```

DEST="$1" ; shift
MEMLIMIT="$1" ; shift

# -x c++:   Explicitly set compile language to C++ (no object
↳ files or
#           other languages autodetected by extension)
# -Wall:   Report all warnings
# -O2:     Level 2 optimizations (default for speed)
# -static: Static link with all libraries
# -pipe:   Use pipes for communication between stages of
↳ compilation
g++ -x c++ -Wall -O2 -static -pipe -o "$DEST" "$@"
exit $?

```

The compilation script has its own command line interface, taking as input the name of the executable to be created, saved inside the variable `DEST`, and the maximum memory usage bound, saved inside the variable `MEMLIMIT`. The result is considered a compilation error if the file with the name of the received executable is not created or if the `compile.sh` script returns a value other than 0.

**testcase\_run.sh** It is another internal DOMjudge script, and it is used to test each test case of a submission. This means that if a certain problem has more than one test case set, this script will be executed multiple times, one per test case. The test of a submission involves firstly running the submission by executing a run script and then comparing the obtained result with the expected output executing another script. It receives as inputs the test case input and expected output as text files, the time limit for the current problem, the directory where to execute the submission in a chroot-ed environment, the absolute paths to the **run** and **compare** scripts and the arguments used by the **compare** script.

**run** Run script called from **testcase\_run.sh**. It is in charge of running the student submission. It receives as inputs a file containing the test input, the path to the file where to write the solution output and the command and options of the program to be run. By default, this is the action performed by the **run** script:

```
exec "$@" < "$TESTIN" > "$PROGOUT"
```

where **TESTIN** contains the path to the test input data, and **PROGOUT** contains the path to the output destination file. The line executes the command specified as input to the script by using **\$@**, which expands to separate the arguments, redirects the test case input to the input of the command, and redirects its output to the destination file. This file can be modified by the users to customize the run phase if a particular problem requires it.

**compare** Executable produced from **compare.cc** to compare the obtained output of the student submission with the current test case's expected output. It parses the output and solution, and as soon as it detects a difference, it calls a specific function in charge of treating it (blank space difference, character difference, expected EOF, ...) and returns wrong answer, or returns accepted solution code otherwise. It also has the possibility of writing to a special **teammessage.txt** file used potentially to write special feedback with regard to the solution. More on this will be seen later. It is also an accessible file to the users to be changed per problem if required.

## 4.2. Running the tools

With the knowledge acquired during the previous phase, we can now delve ourselves into the specifics of the implementation. Our first objective in the process was crucial: deciding where and how the static analyzers and instrumentation will be executed. Any change made to the files described during the previous section will be considered as an alteration of the default DOMjudge behavior, hence, we can say that we are extending it. As we will talk about the structure of the directory of the working environment for this part, it is relevant to explain how it is organized.

The working directory tree looks as shown in Figure 4.2 (files that are not relevant for this section have been excluded). This tree corresponds to a single submission to a problem where students are asked to generate a simple "Hello World!" program. The

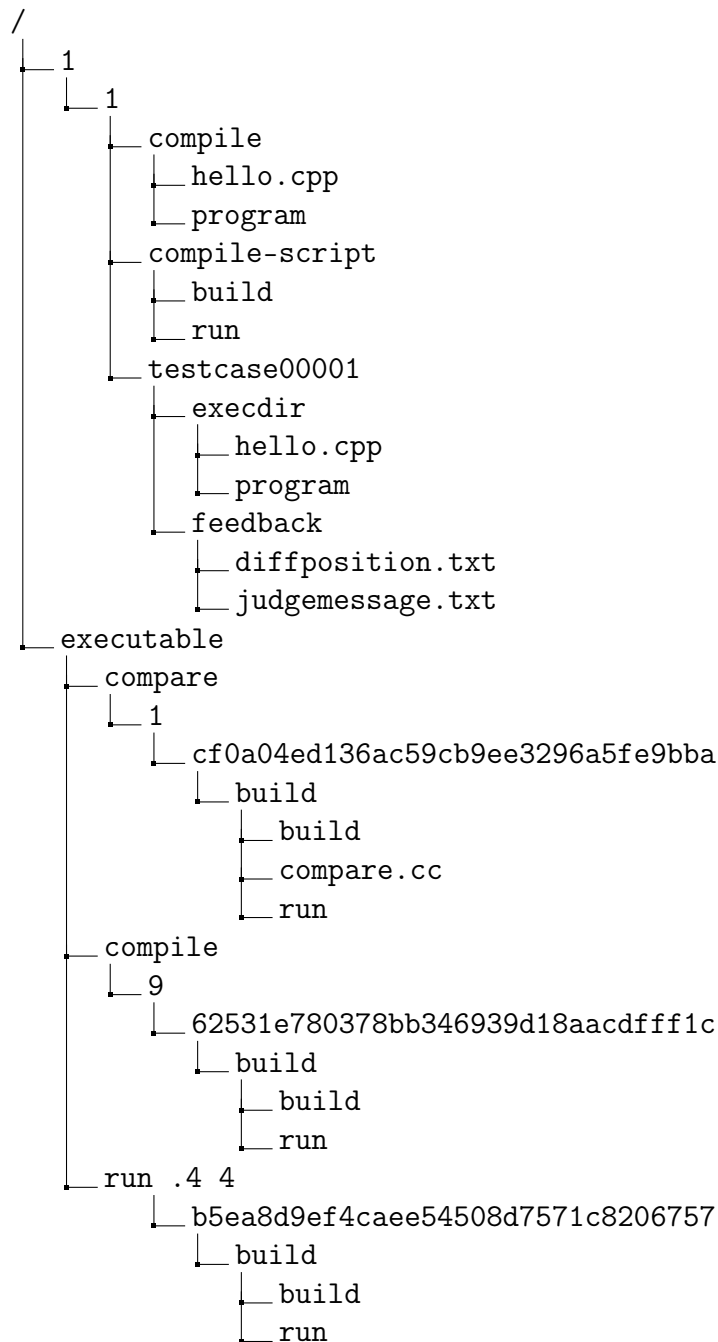


Figure 4.2: Working directory structure of the judgehost

initial node, representing the root directory, with a number indicates the submission identifier. Each subsequent submission made will be enumerated in ascending order. In this folder, the judgehost stores the outputs generated by the compile script. It also contains additional folders for each test case of the problem in the format `testcasexxxx`, enumerated, too, in ascending order. For this specific example, there is a single test case. The output generated by the `compare` executable and `run` script are placed inside it. It is specially relevant to highlight the presence of a “`feedback`” directory, used to provide possible feedback information to the judges

during the compare process. Outside the submission folder, all the necessary files for the different scripts can be found. It would be here where the different scripts would be fetched and placed.

The final tree will be populated with additional files that we included during the implementation process. Having understood how the directory is organized and the submission process, ideas for the implementation begin to flourish. Our initial one involved using the `compile.sh` script for the purpose of running the static analyzers, as it is the place where all submissions are compiled. It could be convenient for the process, as submissions would only be compiled once per execution and we could work from that, since the compilation in this script has to take place anyway. For now, we will introduce how the final version looks like, and later we will discuss some of the alternatives proposed and the reason for them being scrapped or replaced.

The initial phase of the submission process is the compilation of the submitted code. No modifications have been made to this phase, and its behavior has remained untouched. The submission source code will be compiled as per usual and the executable will be made accessible to the run phase.

On the other hand, the run phase has seen substantial changes being made to it. As static analyzers do not require a compilation of the program other than the default one, we can simply run them on the provided submission files. This will take place inside the `run` executable that is called from `testcase_run.sh`. This will generate a parseable file that would then be forwarded to the compare script to generate a message for the team, one per static analyzer.

For the case of sanitizers, we can recompile at this stage, run the program to generate the outputs and then continue with the default run behavior. The main drawback to this approach is that submissions will be compiled twice when sanitizers are enabled. The first compilation of the program takes place during the default compilation phase of the process, and the second compilation takes place when the submission enters the modified run phase. For runs that do not indicate that sanitizers want to be used, these changes will have no effect on the default DOMjudge behavior.

For this reason, we have to allow instructors to customize the type of run that they want to use, as the increase in evaluation time can be an unwanted side effect. To do so, we specified a set of headers that must be included as a first line to the test cases that would adequate the run to the needs. Static analyzers and instrumentation will only be executed if this special headers are contained in the test cases, otherwise, the judge will simply perform the default execution, and it will remain unaffected. The task of reading the headers and identifying what type of run must be performed is delegated to the `run` executable in charge of running the submissions to obtain their outputs. This seemed logical, since the script has direct access to the test cases and is prepared to run compiled code, such as that compiled for the instrumentation. The headers will allow selecting, for instance, if only static analyzers want to be run, or both sanitizers included for our implementation, or even all tools.

How can we, then, indicate what type of run wants to be performed? Headers must be very unlikely to appear accidentally in normal test cases to not be accidentally included, resulting in a non-desirable execution. Also, headers can be specified






	#	sample	download
	 1 	<input checked="" type="checkbox"/>	testcase1.in testcase1.ans
	 2 	<input type="checkbox"/>	testcase2.in testcase2.ans

Figure 4.3: Example test cases for a problem

per test case, this will allow running static analysis on the first test case and address sanitizer on a second one, for example. Or choose only one. Headers are composed of two lines. The first line indicates that a run using analysis tools wants to be executed. For this purpose, we have chosen the Unicode character for *start of heading*:  $s_{0_H}$ . If the test case contains this at the start of it, it will have to be followed with three bits that act as a bit mask to select the tools to be used. The first bit will indicate if we want to use static analyzers, and the second and third one if we want to use undefined behavior sanitizer and address sanitizer, respectively. For instance, if only static analyzers want to be run, we will specify 100; or if both sanitizers want to be run, then we would use 011. By doing this, we allow instructors to choose the specific kind of analysis that has to be performed. The following example represents the content of the test case for a simple “Hello World!” that wants to run static analyzers and UBSan:

```

 $s_{0_H}$ 
110

```

where  $s_{0_H}$  is the representation of the *start of heading* non-printable ASCII control character (whose code is 1) and 110 indicates our tools of choice. Test cases that use these headers have to be included as a test case with sample enabled to be able to show its message to the team, as seen on figure 4.3, where the first test case will output a message, and the second one will not.

This is an undesirable implementation limitation, as marking a test case with sample enabled outputs information regarding its execution time and the output of the submission, information that instructors may not want to show to the students.

The run script will then perform the corresponding action and output it to a file to be able to read it from the compare script. Note that the judge expects for an execution to occur, otherwise it will consider it as a failed result. The way for it to check if an execution has occurred is by looking at the content of the file `program.meta` and checking that the value for the `exitcode` is equals to 0. Aside from that, the file also contains all the information in regard to the program’s time performance. This is the content of the file:

```
memory-bytes: x
exitcode: x
wall-time: x
user-time: x
sys-time: x
cpu-time: x
time-used: cpu-time
time-result: x
output-truncated: x
stdin-bytes: x
stdout-bytes: x
stderr-bytes: x
```

Note that `x` acts as a placeholder for the actual values. A workaround for avoiding this execution is to write manually into `program.meta` that the exitcode was 0, but by doing so we would miss all the other relevant information that is written into the file. So, for the particular case of the sanitizers, we decided to leave this default execution, so they will have to be executed twice, once inside the run executable to obtain the generated sanitizer output using the special compiled file, and another one that follows the default judge behavior.

The `compare` executable should also be able to identify if we are currently judging the special case run. For that, the `testcase_run.sh` script writes the header options to a separate `runtype.info` file inside the compare phase working directory. This is not a strictly necessary step, as `compare` has access to the input test case file, but makes the process simpler. The `compare` will be able to access the information on this file and act accordingly.

With the above changes, instructors will have the freedom to choose what kind of run they want to perform, the code will remain unaffected, compilations for static analyzers will only occur once and the options are easily scalable; the drawbacks to this process are the double compilation that occurs during the instrumentation runs and the time restrictions imposed by the analysis. The output of this process has to be shown to the students.

### 4.2.1. Implementation alternatives

Before the final implementation could take shape, we had to perform some iterations on it and test alternatives to find the best possible one. As mentioned during the previous section, our initial idea involved modifying the `compile.sh` script. This process would involve including the execution of the static analyzers at some point during the compilation phase and creating new files to store the output of the static analyzers for generating the feedback messages later. The `compile.sh` script generated an `SAoutput.txt` file with the content of the static analyzers output. From `testcase_run.sh` we would then have to move this file to the feedback directory to make it accessible from the `compare` script, place where the message would be generated following the same principle of running parser scripts that we ended up using. What remained would be to open the file in `compare` phase and transfer the

`SAoutput.txt` content to a `teammesssage.txt` file.

We wanted to play around with other possibilities following this same principle, like passing the submission file(s) itself to the `compare` script and executing the static analyzers there, which is also possible. To achieve this, we can pass as an additional argument to the `compare` script the name of the source file, to know the name that `compare` should read. We modified a bit how the call was made.

Despite both options explained above being functional, we later realized that it is not an ideal solution by far. We not only modified what is being passed to the scripts in this second iteration, it is also repeated every time for problems with multiple test cases, producing the same output for all of them and performing the analysis every time, which is not optimal and would result on a slower execution. In addition, if a compilation is done with instrumentation and multiple test cases exist, it would result in a much slower execution.

We ended up discarding the possibility of using a secondary `compare` and run scripts for the analysis, too, since a `compare` script can be specified per problem, but not per test case, which would result on the same code being executed for every test case; again, degrading the performance. It would also be less convenient for the instructor to have to specify a `compare` script per problem.

Another contemplated possibility, still dependent on the compilation script, was finding a way to indicate, somehow, the type of compilation that wants to be performed. The idea behind this was that the compilation with instrumentation would only be performed when indicated. For this, we would have to forward the information that indicates this to the `compile.sh` script. It did work for problems that had a single test case, compilation would be performed with instrumentation when indicated. The problem arose when more than one test cases were included and each requires a different type of compilation. It is at this stage that the judgehost detects the fact that the resulting compilation results differ from a previously obtained one and triggers an error. Since this behavior, under normal circumstances, could indicate that there is a malfunctioning machine, domserver disables the judgehost that triggered the warning.

The above described options became valuable insights into what would later become the final implementation. Exploration of these aspects allowed us to identify the most appropriate methods to focus on.

## 4.3. Providing feedback

Having now an appropriate method to run the static analyzers and instrumentation, what remains is to show the feedback to the students. To do this, we show it in the same window used to communicate the output of the compiler once a submission had been evaluated by the judge.

The file in charge of formatting this output is the HTML template *submission.html.twig*. Modifying this template to allow for an additional text box with a message is very simple, and the content that we provide to this new text box is that contained in a file called `teammesssage.txt`. By default, if an existing `teammesssage.txt` file is present, the template will display it underneath the sec-

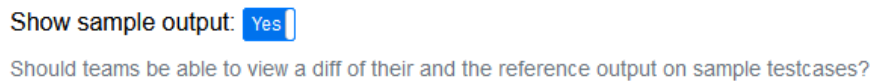


Figure 4.4: Option enabled from DOMjudge settings

tion of the test case run that generated it, but we modified this behavior, as the file is currently not being used by the judge. Information regarding this matter was given to us through an email thread with the developers, as for how it stands at the moment of developing this project, it has not yet been included into a main release of DOMjudge, hence it does not appear in its documentation.

Anything written during the compare stage to this file will be shown to users of the judge through the text box. The message will contain the parsed output generated for the static analysis and instrumentation, if used. Essentially, it will display the concatenation of all the generated team messages. Unfortunately, this removes the possibility of having a message being shown per test case run. It is a design decision that we would have preferred to avoid, but we consider that having a single message being shown is more convenient for this specific use case than separating it per run.

To be able to activate the displaying of messages, a special option must be enabled in the DOMjudge settings as an administrator. When enabling "Show sample output" under "General settings"(see figure 4.4), instructors will be able to use the described features. Again, we would have preferred this feature to not be linked to this option, but the DOMjudge version being used detects when this option is enabled to display the contents of `teammessages.txt`.

### 4.3.1. Obtaining the number of tries

In order to make the displayed messages more interesting and informative, we decided that it would be a good idea to know the number of attempted submissions for a student/team for a given problem. This would allow us to create a more generic message for the first attempts, hinting as to where the problem could originate from if a submission is incorrect, and guide the students further as the attempts increase. We do not want to give a solution to the problem straight away, but at some point we must guide the student somehow if they cannot find a fix.

The number of attempts for a problem can be obtained making an API request to obtain the scoreboard, for what the identifier of the contest is needed; this generated a series of problems involving modifications that had to be applied to the database schema in order to make it accessible inside `judgedaemon.main.php`. The documentation includes all the available methods of the API in an organized manner. [8, 37, 13] The first part to obtaining the number of tries, involves making the required API request from `judgedaemon.main.php` to obtain the specific contest. The structure of the returned JSON file for this call is as follows:

```
{
```



```

"event_id": "string",
"time": "string",
"contest_time": "string",
"state": {
  "started": "string",
  "ended": "string",
  "frozen": "string",
  "thawed": "string",
  "finalized": "string",
  "end_of_updates": "string"
},
"rows": [
  {
    "rank": 0,
    "team_id": "string",
    "score": {
      "num_solved": 0,
      "total_time": 0,
      "total_runtime": 0
    },
    "problems": [
      {
        "label": "string",
        "problem_id": "string",
        "num_judged": 0,
        "num_pending": 0,
        "solved": true,
        "time": 0,
        "first_to_solve": true,
        "runtime": 0,
        "fastest_submission": true
      }
    ]
  }
]
}

```

We are interested in the value contained as a pair of the “num\_judged” field. It represents the number of judged submissions for a problem so far, so if the current attempt is the first one, the value retrieved will be 0, and vice versa. To retrieve it, we have to iterate through the teams of a given contest to obtain the current team and then through the problems that have been attempted by the team.

What remains is to forward this information to the `testcase_run.sh` script, which makes it available inside a dedicated file to the `compare` executable. It was done this way to avoid modifying what the `compare` executables receive as arguments, in case a customized `compare` is decided to be used following the default DOMjudge conventions.

This is the process that involves retrieving the attempts, and we needed the contest identifier to do so; as already said, the API method receives a contest identifier to return the scoreboard. At first, we did not have direct access to this value, being our first idea to obtain it to retrieve the whole list of contests and iterate through it until we found the correct one. A much better solution was provided to us thanks to the team, that is, modifying the `JudgeTask` entity class to add an additional field containing this information. `judgedaemon.main.php` includes the `JudgeTask` entity class, which is used to keep track of the individual judgetasks - `judgedaemon.main.php` is called once per judging task. It uses this entity class to obtain a number of different values to make requests to the API; the submission ID, for instance.

To allow for these changes to be applied, we made modifications to the `JudgeTask` entity class in order to add a `contest_id` property index and column to be accessed from `judgedaemon.main.php`. In order to successfully update the contest identifier, we included a line in the function in charge of adding the current submission identifier to also add the current contest identifier to the entity:

```
public function getSubmitid(): ?int
{
    $this->contestidjt =
        ↪ $this->submission?->getContest()->getCid();
    return $this->submission?->getSubmitid();
}
```

This line gets the current contest from the submission and assigns its id to the `JudgeTask` entry.

Next, we had to go to the `Symfony`<sup>6</sup> directory and execute the appropriate commands in order to migrate the database, that is, reflect the changes made to `JudgeTask` on the actual database. The base directory for `Symfony` is located at `/webapp` in the `DOMserver`. All database migrations are located inside the directory `webapp/migrations`. A new file was generated in this directory with the corresponding change to the database schema.

Some debugging was required during this process to understand some other changes that had to be made to allow this to happen. The most relevant change was modifying the file `.env.local`, adding an updated environment variable value that is required by `Symfony` as per their documentation.<sup>7</sup> The file now contained

```
DATABASE_URL=mysql://DOMjudge:djpw@127.0.0.1:3306/DOMjudge?serve_
↪ rVersion=mariadb-10.11.6
```

This line specifies the database URL, with the database user, password, host and port number, as well as the database used and its version in the following format:

---

<sup>6</sup>Symfony is an open-source PHP framework. It is used to develop software that simplifies the web design process <https://builtin.com/software-engineering-perspectives/symfony> [41]

<sup>7</sup><https://symfony.com/doc/current/doctrine.html> [29]

```
mysql://<user>:<password>@<host>:<port>/<database>?serverVersion_
↪   =<version>
```

All the fields can be found in `etc/dbpasswords.secret` except for the version, which is obtained by executing `mysql -V` in the server container. Once this was applied, we could successfully make the desired migration, reflecting the changes in the database.

### 4.3.2. Generating the message

The last step to forward a complete message is actually parsing the static analyzers output and putting the pieces together. This process is carried out from the `compare` script, as it is now the latter stage for all the previous processes. It has access to the number of attempts, the static analyzers output, and the `teammesssage.txt` file that is later used to forward the message to the students.

To achieve this, we decided that the best approach in our case would be to reuse part of the Python scripts that we already had developed for the analysis in section 3.2, and make the appropriate call from the `compare` script to run these parser scripts. The first step was including the new scripts in DOMjudge, and make sure that they are included inside our image. This was done by adding them inside the same directory as the `testcase_run.sh` script and adding them to the Makefile (`judge/Makefile`).

```
install-judgehost:
    $(INSTALL_PROG) -t $(DESTDIR)$(judgehost_libjudgedir) \
        compile.sh build_executable.sh testcase_run.sh
    ↪   chroot-startstop.sh \
        check_diff.sh evict version_check.sh python_script.py
```

What remains now is to forward these scripts to the `compare` working directory, which is simply achieved from the `testcase_run.sh` file as it has access to it. Each script will only be executed if the option that selects their respective analysis tool is indicated in the header of a test case. They will parse the output file generated by the tool and output a message to the `teammesssage.txt` file. For instance, the script used to generate the message for ClangTidy is called `parse_clang.py`, and the file that it expects as input is called `clangtidy.yaml`, which is created when Clangtidy runs its analysis on the source code of the submission.

## 4.4. Preparations and Setup

We have explained everything with regard to the implementation up to this point, but not how they were applied. All the changes were tested in a functional DOMjudge environment. Diving a bit into all the DOMjudge offerings and repositories, we discovered that the developers maintain a Docker<sup>8</sup> image with all the require-

<sup>8</sup>Docker is an open platform for developing, shipping, and running applications. It provides the ability to run an application in an isolated environment called a container [18].

ments in order to deploy a functional DOMjudge locally. It is specially intended for those that want to collaborate with the project and need a testing environment, so it worked perfectly; with an appropriate use of it, it allowed us to have the same working environment replicated on all of our machines. These images are available in the public Docker repository, Docker Hub, as `DOMjudge/domserver` and `DOMjudge/judgehost`.

Before being able to start the containers for these images, an instance of a MariaDB database must exist beforehand. Fortunately, it is very easy to set up using Docker, and the Docker Hub page of the DOMjudge itself explains the steps that must be followed to deploy it, granting the commands that must be executed.

In order to be able to run these containers locally, Docker is required to be installed in the machines. An important aspect to consider is that, as previously mentioned, DOMjudge uses Linux Control Groups or cgroups for process isolation in the `judgedaemon`. So, to enable cgroup requirements for DOMjudge, we have to add cgroup memory and swap accounting to the boot option. Additionally, modern Linux distributions have cgroup v2 enabled by default, and we have to specify that we want to boot the host into cgroup v1<sup>9</sup>, which is the only one supported by DOMjudge. All the previously mentioned can be achieved by adding the following in `/etc/default/grub`:

```
GRUB_CMDLINE_LINUX_DEFAULT="quiet cgroup_enable=memory
↪ swapaccount=1 isolcpus=2 systemd.unified_cgroup_hierarchy=0"
```

The content of this file indicates to the system bootloader at system boot time what arguments have to be passed to the system kernel to configure the system. A bootloader is a program which enables the selection of the installed operating system or kernel to be loaded at system boot time. It also allows the user to pass arguments to the kernel<sup>10</sup>. So, the code in the previous step is executed during the boot process to apply the DOMjudge requirements: booting into cgroup v2. In addition to this, cgroup memory and swap accounting has also been added as per the recommendations in the documentation [36].

The key `GRUB_CMDLINE_LINUX_DEFAULT` lists commandline arguments to add only to the default menu entry. [11] Commandline arguments are additional inputs forwarded to a command in order to specify files to used, options to be enabled, or any other parameter required for a command to execute properly. Many are optional, and are used to specify a behavior for a command other than the default one. [38]

#### 4.4.1. Initializing the containers

Having this now configured, we can initialize the different containers in order to have a functional DOMjudge installation. The first step in the process involves

<sup>9</sup><https://www.DOMjudge.org/docs/manual/main/install-judgehost.html#linux-control-groups> [36]

<sup>10</sup>[https://access.redhat.com/documentation/en-us/red\\_hat\\_enterprise\\_linux/5/html/installation\\_guide/s1-grub-what-is](https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/5/html/installation_guide/s1-grub-what-is) [20]

initializing the database, since the domserver requires that it exists beforehand. This can be achieved by running the commands contained in the guide that the DOMjudge team has developed. By default, the configuration exposes the server on port 13306 on the local machine.

Once the database is created, a domserver instance can be started, indicating that we want to use the existing running database to store information. Initially, the domserver will have no user aside from the administrator account, and no problems or contests available.

Lastly, as many judgehosts as the installation requires can be initialized. Judgehosts will be linked to the domserver instance and receive a different hostname for each.

This is everything necessary in order to have a functional DOMServer ran using Docker. The web interface can be accessed on `http://localhost:12345/`, allowing only for admin login at first, as no other account is provided by default. Files can be edited directly from the running container by starting an interactive shell session. A main consideration is that the judge at this stage lacks additional packages that could be necessary in order to make effective modifications, such as Vim as an editing software. They have to be installed in order to make them available to be used. Other packages and dependencies required are those needed in order to run the static analyzers, Clang-Tidy and Cppcheck, as GCC comes preinstalled. Installing a text editor directly, for example, is straightforward, as it can be directly installed through the interactive shell session, along with any other desired package. In order to be able to run the static analyzers, we have to make them accessible from the chroot environment, as submissions are evaluated and executed there. To do so, we can execute the script `bin/dj_run_chroot`, which runs an interactive shell or a command inside the chroot. Since apt is available inside the chroot, we can install the required packages using it. The name of the packages that have to be installed in order to make the static analyzers usable are: `clang-tidy` and `cppcheck`, and the following command installs them:

```
apt-get update && apt-get install -y clang-tidy cppcheck
```

This is not the most convenient way to do so, and it can be configured to install everything required beforehand, but it will suffice for our testing purposes at this stage.

#### 4.4.2. Building new images

Early in the process of applying our changes, we realized that certain features that were available in the current GitHub main branch of DOMjudge were not yet available in the latest release. Among these features, there was the feedback message options, that contained all the changes related to the `teammessage.txt` file that we used for our implementation. This meant that the Docker containers that we had been using up to that point did not have this feature, as they are based on the latest available stable release of DOMjudge. The implementation of it has been discussed

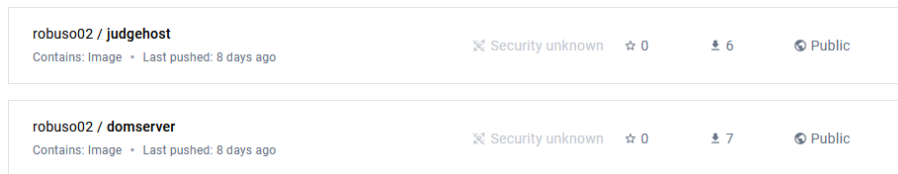


Figure 4.5: DockerHub repositories

in a GitHub issue with the developers before,<sup>11</sup> and the feature was finally included on pull request #2016.<sup>12</sup> Therefore, we had to generate a new image from GitHub’s source code to be able to use it.

Fortunately, the DOMjudge developers maintain a `DOMjudge-packaging` repository on GitHub as part of the project. It contains all the requirements to create Docker images which can be used to run DOMjudge, provided as part of their support for contributors. We decided not to use it at first because we could simply deploy the published Docker images, and it worked just fine for us at first glance. By default, the provided scripts set everything up with maintained versions that do not contain the latest features that we need, being it the same one that is obtained during earlier Docker setups. To allow for the use of customized versions, we had to make some changes to the scripts and download the `src` code directly from the repository and run `make dist`.<sup>13</sup>

An advantage that this approach also provided is that we could now specify what packages we want to include in the process of creating the image to be used, and not have to install them manually every single time. We removed them later for the final version, as they are no longer required.

An important note to make is that the latest Docker version release has features that are incompatible with the build process, and we had to downgrade to Docker version 24.0.6. After all the changes were made, we successfully generated new images to be used for our testings. We uploaded them to a public Docker Hub repository in order to make it accessible to all the members.

Apart from the team messages inclusion, this version comes with a more refined looking UI and additional features that we would not use for the development of the project, but are either way nice to have.

### 4.4.3. Automating the process

At this stage, we decided that it would be very convenient to have an appropriate automated method to deploy the environment on all of our local machines without the hustle of doing it manually. The logic path to follow was the one of creating a Docker compose file [17].

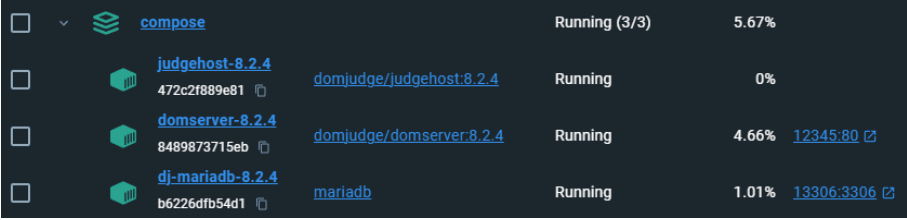
The steps to carry this out involved the creation of a script that orchestrates

<sup>11</sup><https://github.com/DOMjudge/DOMjudge/issues/991> [9]

<sup>12</sup><https://github.com/DOMjudge/DOMjudge/pull/2016>

<sup>13</sup>A makefile is a way of automating software building procedure and other complex tasks with dependencies. The `dist` rule in the generated Makefile can be used to generate a gzipped tar file and other flavors of archive for distribution. [10] [26]

the process. The script executes the appropriate `docker-compose` commands on two compose files. The first one creates the MariaDB database and DOMServer containers, and the second one creates the one for the judgehost, and they are created in the mentioned order, as the judgehost depends on the DOMServer, which depends on the database. In the end, we obtain the same result but faster and in a simpler manner. Figure 4.6 displays the generated containers using the compose files. It successfully automates the container generation process.












<input type="checkbox"/>	▼	 <b>compose</b>		Running (3/3)	5.67%
<input type="checkbox"/>		 <b>judgehost-8.2.4</b>	472c2f889e81 	<a href="#">domjudge/judgehost:8.2.4</a>	Running 0%
<input type="checkbox"/>		 <b>domserver-8.2.4</b>	8489873715eb 	<a href="#">domjudge/domserver:8.2.4</a>	Running 4.66% <a href="#">12345:80</a> 
<input type="checkbox"/>		 <b>dj-mariadb-8.2.4</b>	b6226dfb54d1 	<a href="#">mariadb</a>	Running 1.01% <a href="#">13306:3306</a> 

Figure 4.6: Containers view from Docker Desktop





# Chapter 5

## Real use case scenario

For testing and showcasing the performance of the modified DOMjudge, we prepared two examples that serve as a proof of concept. Firstly, a test problem and submission that contains a variety of fictitious errors to be detected during the evaluation process that will serve as a showcase of the analysis tools outputs. For the second example, we simulated a fictitious student submission that attempts to be as realistic as possible.

### 5.1. Showcase

We have developed a minimal artificial problem that expects the following output:

```
Correct solution!
```

and a `testcase.cpp` file that will be used as the submission. This submission file will trigger several warnings that will be detected by the static analyzers and the sanitizers. For this purpose, we will indicate as a header of the test case that we want to perform a run with all the included features. This is the content for the test case submission:

```
[...] // omitted lines
void * p;

int main() {
    int ways[3];
    for (int i = 0; i < 3; i++)
    {
        ways[i] = i;
    }
    ways[2] = ways[2]; //selfAssignment

    if (ways[2] == 2);
```

```

{
    ways[1] = 7;
}

p = malloc(7);
p = 0;

int j = numeric_limits<int>::max();
j += 1;

cout << "Incorrect solution with all kinds of errors" <<
    << endl;

return 0;
ways [1] = ways[0]; //unreachableCode
}

```

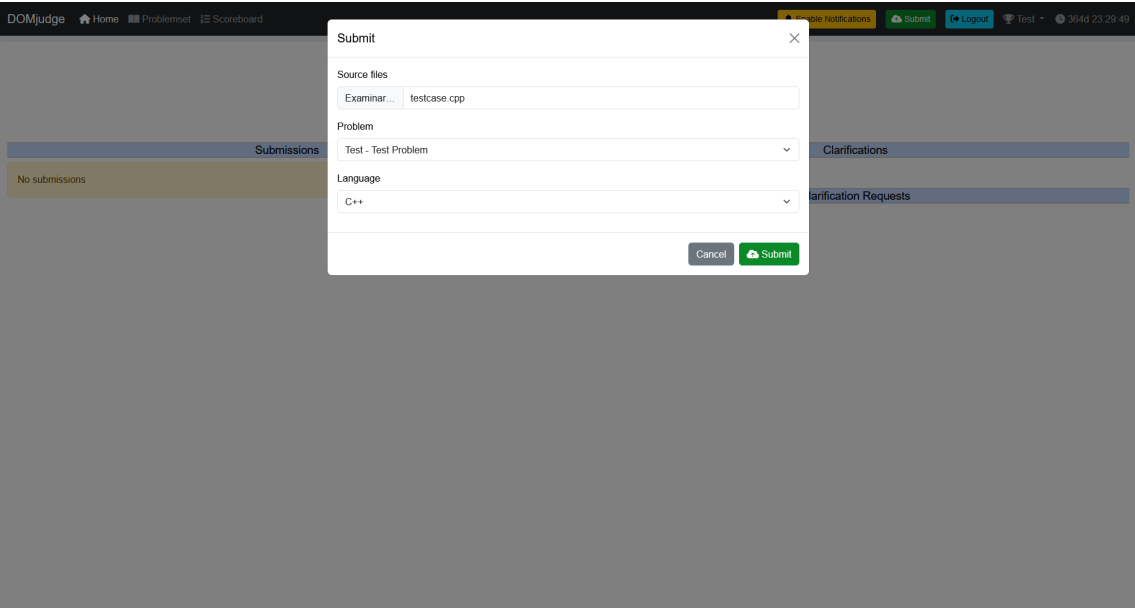
As usual in DOMjudge, we will be greeted with an empty page, waiting to receive a submission.

The screenshot shows the DOMjudge web interface. At the top, there's a navigation bar with links for Home, Problemset, and Scoreboard. The Scoreboard section displays a table with columns RANK, TEAM, SCORE, and TEST. The table shows one entry: rank 1, team 'prueba', score 0, and test 0. Below the scoreboard, there are three main sections: Submissions (showing 'No submissions'), Clarifications (showing 'No clarifications'), and Clarification Requests (showing 'No clarification request' and a 'request clarification' button).

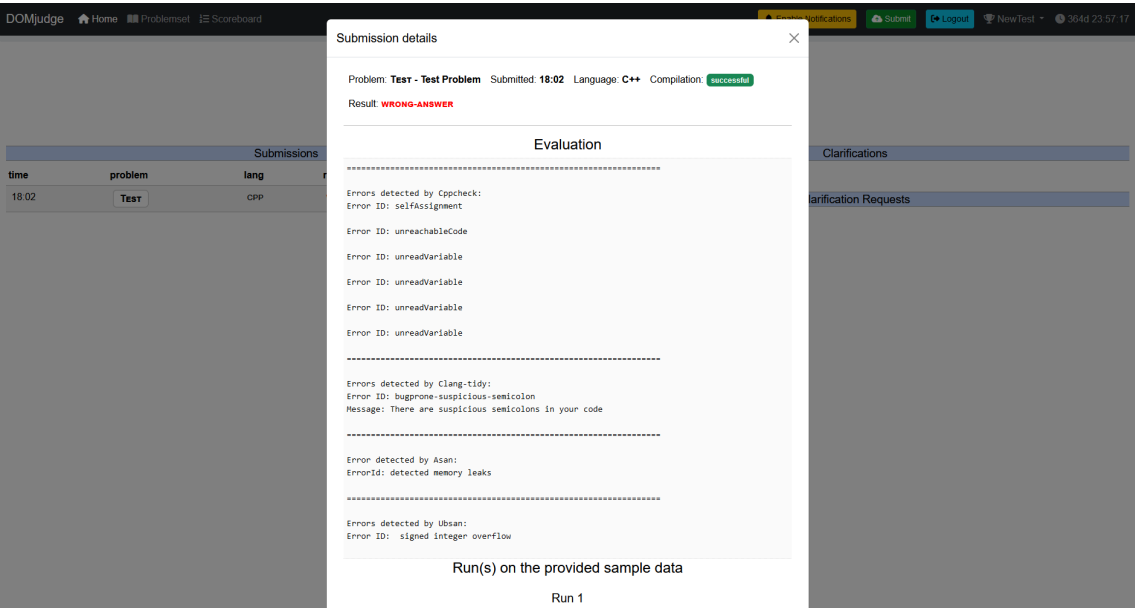
The first step is to make the submission to the judge. The input file will output the following text:

```
Incorrect solution with all kinds of errors
```

this will be assessed as *Wrong Answer*. Once the file is submitted, a judgehost will get the submission from the queue and start processing it. As we have indicated that we want to run static analyzers and sanitizers in the header of the test cases, the judgehost will use them.



The submission will take some time to evaluate, as both sanitizers are being added to the compilation. Once the verdict is returned, we can see the generated feedback detailed.



```
=====

Errors detected by Cppcheck:
Error ID: selfAssignment

Error ID: unreachableCode

Error ID: unreadVariable

Errors detected by Clang-tidy:
Error ID: bugprone-suspicious-semicolon
Message: There are suspicious semicolons in your code

Error detected by Asan:
ErrorId: detected memory leaks

Errors detected by Ubsan:
Error ID: signed integer overflow

Run(s) on the provided sample data

Run 1
```

```
Error ID: unreadVariable
```

```
Error ID: unreadVariable
```

```
Error ID: unreadVariable
```

```
=====
```

```
Errors detected by Clang-Tidy:
```

```
Error ID: bugprone-suspicious-semicolon
```

```
Message: There are suspicious semicolons in your code
```

```
=====
```

```
Error detected by ASan:
```

```
ErrorId: detected memory leaks
```

```
=====
```

```
Errors detected by UBSan:
```

```
Error ID: signed integer overflow
```

As this is our first attempt, we will receive a very vague description of some issues found. The outputs for each kind of evaluation tool has been separated from one another for demonstration purposes, this will not be shown to the students. We now upload a second time, making, some changes to the code in order to, hopefully, fix the issues. In our case, we will upload the same file to trigger the same warnings and inspect how the returned message changes.

```
=====
```

```
Errors detected by Cppcheck:
```

```
Error ID: selfAssignment
```

```
Location of the error: Line: 16, Column: 16
```

```
Error ID: unreachableCode
```

```
Location of the error: Line: 33, Column: 5
```

```
Error ID: unreadVariable
```

```
Location of the error: Line: 20, Column: 20
```

```
Error ID: unreadVariable
```

```
Location of the error: Line: 26, Column: 11
```

```
Error ID: unreadVariable
```

```
Location of the error: Line: 27, Column: 7
```

```

Error ID: unreadVariable
Location of the error: Line: 33, Column: 17

=====

Errors detected by Clang-Tidy:
Error ID: bugprone-suspicious-semicolon
Message: Be careful with if/while/for with a semicolon after the
→ condition

=====

Error detected by ASan:
ErrorId: detected memory leaks

=====

Errors detected by UBSan:
Error ID: signed integer overflow
Location of the error: Line: 27, Column: 7

```

After the second attempt, we see that the generated message now also includes the locations where the triggered warnings were found in some cases, or a more detailed message on another ones. We will now upload the submission for a third time, which will trigger a fully detailed output for the triggered warnings.

```

=====

Errors detected by Cppcheck:
Error ID: selfAssignment
Message: Redundant assignment of 'ways[2]' to itself.
Location of the error: Line: 16, Column: 16

Error ID: unreachableCode
Message: Statements following 'return' will never be executed.
Location of the error: Line: 33, Column: 5

Error ID: unreadVariable
Message: Variable 'ways[1]' is assigned a value that is never
→ used.
Location of the error: Line: 20, Column: 20

Error ID: unreadVariable
Message: Variable 'j' is assigned a value that is never used.
Location of the error: Line: 26, Column: 11

```

```

Error ID: unreadVariable
Message: Variable 'j' is assigned a value that is never used.
Location of the error: Line: 27, Column: 7

Error ID: unreadVariable
Message: Variable 'ways[1]' is assigned a value that is never
→ used.
Location of the error: Line: 33, Column: 17

=====

Errors detected by Clang-Tidy:
Error ID: bugprone-suspicious-semicolon
Message: Be careful with if/while/for with a semicolon after the
→ condition as it can make the following code behave
→ unexpectedly: it can execute unconditionally or execute just
→ once

=====

Error detected by ASan:
ErrorId: detected memory leaks

=====

Errors detected by UBSan:
Error ID: signed integer overflow
Message: signed integer overflow: 2147483647 + 1 cannot be
→ represented in type 'int'

Location of the error: Line: 27, Column: 7

```

For this attempt, we receive the a detailed reason for the warnings, since the previous clues were not enough. If this is the case and we now upload a file that outputs the correct expected result, it will be accepted by the judge.

```

=====

Errors detected by Cppcheck:
Error ID: selfAssignment
Message: Redundant assignment of 'ways[2]' to itself.
Location of the error: Line: 16, Column: 16

Error ID: unreachableCode
Message: Statements following 'return' will never be executed.

```

Location of the error: Line: 33, Column: 5

Error ID: unreadVariable

Message: Variable 'ways[1]' is assigned a value that is never  
→ used.

Location of the error: Line: 20, Column: 20

Error ID: unreadVariable

Message: Variable 'j' is assigned a value that is never used.

Location of the error: Line: 26, Column: 11

Error ID: unreadVariable

Message: Variable 'j' is assigned a value that is never used.

Location of the error: Line: 27, Column: 7

Error ID: unreadVariable

Message: Variable 'ways[1]' is assigned a value that is never  
→ used.

Location of the error: Line: 33, Column: 17

=====

Errors detected by Clang-Tidy:

Error ID: bugprone-suspicious-semicolon

Message: Be careful with if/while/for with a semicolon after the  
→ condition as it can make the following code behave  
→ unexpectedly: it can execute unconditionally or execute just  
→ once

=====

Error detected by ASan:

ErrorId: detected memory leaks

=====

Errors detected by UBSan:

Error ID: signed integer overflow

Message: signed integer overflow: 2147483647 + 1 cannot be  
→ represented in type 'int'

Location of the error: Line: 27, Column: 7

Notice that, despite receiving a correct verdict, some messages are still shown to the students, as there are other performance or style concerns that may be addressed even with an accepted solution, the instrumentation and sanitizers will still be able to detect errors in the code if that is the case. This behavior is beneficial, as users

will still be able to know if there is something to improve in their codes.

## 5.2. Realistic Case Scenario

For the realistic case scenario, we used a submission that attempts to be as close to reality as possible. To simulate real behavior, every iteration of a submission will apply a fix from the suggestions. The submission is attempting to solve a problem presented as follows:

*There is a celebration at a table in the shape of a ring where  $n$  guests can sit inside and another  $n$  guests outside (with  $n \leq 10,000$ ). The sequence of guests inside and the sequence of guests outside are fixed, but it is still possible to decide how to align the two sequences. Each person has a level of enthusiasm (an integer whose absolute value is less than  $10^6$ ), and the enthusiasm of the celebration is the sum of the products of the enthusiasms of the people who are facing each other at the table. The problem consists of finding the maximum possible enthusiasm.*

The solution of the problem is this formula where  $v_1$  and  $v_2$  are the enthusiasm arrays for the inside and outside sequences of guests:

$$\max\left\{\sum_{i=0}^{|v_1|-1} v_1[i] \cdot v_2[i+k] \mid 0 \leq k \leq |v_1|-1\right\}$$

Two test cases are included for the problem. The first one will be flagged with the header used to apply all the tools simultaneously. Since the output of a static analyzer is independent of the test case input, we will indicate for the second test case that we only want to run the instrumentation. This will avoid including the same errors and warnings twice for the case of the static analyzers.

Let us submit first a faulty solution whose more relevant function is the following `sumProduct`:

```
using tAnim = int;

tAnim sumProduct(vector<tAnim> v1, vector<tAnim> v2, int offset)
{
    int numSeats = v1.size();
    tAnim total = 0;

    for (int i = 0; i < numSeats; ++i)
        total += v1[i] * v2[offset + i]; // Array out of bounds
        ↪ access

    return total;
}
```

Once it is submitted, we get the following generated output:



Submission details

×

Problem: **CELEBRATION - Celebration** Submitted: **14:59** Language: **C++** Compilation: **successful**

Result: **WRONG-ANSWER**

Evaluation

Message: Try avoiding operating between different types as it can cause unexpected behavior  
A heap overflow was detected, this is usually due to an out of bounds array access but can a

Error: signed integer overflow

Error: signed integer overflow

A heap overflow was detected, this is usually due to an out of bounds array access but can a

Run(s) on the provided sample data

Run 1

Description	
Runtime	0.002 sec
Result	WRONG-ANSWER

Program output

175  
257

Error output (info/debug/errors)

There was no stderr output.

Run 2

Description	
Runtime	0.001 sec
Result	WRONG-ANSWER

Program output

A heap overflow was detected, this is usually due to an out of

- bounds array access but can also be caused by repeated or
- big memory allocations using the malloc function or the new
- keyword.

Error: signed integer overflow

Error: signed integer overflow

```
A heap overflow was detected, this is usually due to an out of
→ bounds array access but can also be caused by repeated or
→ big memory allocations using the malloc function or the new
→ keyword.
```

containing the error:

```
A heap overflow was detected, this is usually due to an out of
→ bounds array access but can also be caused by repeated or
→ big memory allocations using the malloc function or the new
→ keyword.
```

hinting at an array out of bounds access. This clue may help us find where the bug is. In this example, there is a missing modulo to never exceed the maximum number of available seats. After fixing it, we submit the code and obtain a new result.

```
Message: Try avoiding operating between different types as it
→ can cause unexpected behavior
```

```
Error: signed integer overflow
```

```
Error: signed integer overflow
```

In this second attempt, there is no longer an out-of-bounds access to the array, but we can still see the other same message and errors as in the previous attempt since we have not addressed them. No location for the errors are being shown yet. The same error appears twice since it is occurring at two different locations, but we do not know them yet. We will make another submission trying to fix the issues.

```
Message: narrowing conversion from 'std::vector::size_type' (aka
→ 'unsigned long') to signed type 'int' is
→ implementation-defined, this is best avoided as it can lead
→ to unexpected behavior
```

```
Error: signed integer overflow
```

```
Location of the error: Line: 15, Column: 18
```

```
Error: signed integer overflow
```

```
Location of the error: Line: 15, Column: 9
```

Unfortunately, the problems were not solved, but we now have the location of the errors and a more detailed description message. Inspecting the “signed integer overflow” error, we can see that it is found at line:

```
total += v1[i] * v2[(offset + i) % numSeats];
```

The reason behind this error trigger is that the value generated by the product is too large to fit inside the current 32 bit integer variable that is used to store it ( $(10^6)^2 > 2^{31}$ ). Execution is not stopped despite this. It is something that should be addressed, as it can lead to undefined behavior. Observing the returned message, we are determined to make some changes in order to achieve a better solution in the next submission. The error is caused because of the size of the type used for the alias declaration.

```
using tAnim = int;
```

The newly modified submission will fix it by specifying that we want to use 64-bit integers, greater than the 32-bit integer used before.

```
using tAnim = int64_t;
```

The generated team message is still shown even when the submission is evaluated as correct, as mentioned earlier. This will allow to be able to detect possible performance issues even when reaching a right solution that compute the correct result. With this in consideration, a new attempt is uploaded.

```
Message: narrowing conversion from 'std::vector::size_type' (aka
↳ 'unsigned long') to signed type 'int' is
↳ implementation-defined, this is best avoided as it can lead
↳ to unexpected behavior
Message: narrowing conversion from 'tAnim' (aka 'long') to
↳ signed type 'int' is implementation-defined, this is best
↳ avoided as it can lead to unexpected behavior
```

We will attempt to change the code once more to try to eliminate the warnings and get the correct solution by changing the type of the variable `sum` used inside the function `solvecase()` from `int` to `tAnim`:

```
vector<tAnim> inner(numSeats), outer(numSeats);
```

This is the returned message for the submission:

```
Message: narrowing conversion from 'std::vector::size_type' (aka
↳ 'unsigned long') to signed type 'int' is
↳ implementation-defined, this is best avoided as it can lead
↳ to unexpected behavior
```

We did get an *accepted* verdict this time. Still, the warnings are displayed. It is common to have some despite writing proper code, as many different checks are made. With this test, we have tested the correct performance of the modified judge on a realistic student submission. Errors can sometimes be puzzling, especially when specialized language is used to describe them. In our approach, we have attempted to ease the readability of the warnings.



## Conclusions and Future Work

### 6.1. Discussion and conclusions

In this thesis we have made a study on two different static analyzers (Clang-tidy and Cppcheck), we have analyzed the outputs these tools gave us in a dataset of more than 4000 submissions. After the analysis, we have concluded which checks were more relevant. Then, we followed a similar process to understand instrumentation, we studied ASan and UBSan. While those studies were being made, we also started modifying the DOMjudge platform, in order to develop a way to communicate to the student the feedback we were creating. Finally, as we finished all those tasks, we developed a didactic, personal feedback based on the diagnostics issued for the student submissions. This feedback is shown to the student via `teammesssage` (a functionality of DOMjudge) when they submit their code.

After fixing Clang-Tidy, Cppcheck, ASan, and UBSan as target tools, we analyzed their diagnostics. There are no known studies about these tools on student submissions. So in this thesis we have made an exhaustive analysis on all the diagnostics that could be raised in student submissions and made clear which diagnostics are we are actually going to communicate, excluding those with little to no relevance for a didactic and useful feedback.

We then integrate the tools into DOMjudge. To make it, we have requested aid from the developers of the platform in order to fix various issues that arose during the pertinent modifications of the platform (for example to include an “outdated” messaging system that the platform provided “teammesssage”) and changes to the assessment process of the online judge, specifically, the `compare` and `run` scripts in order to run the static and dynamic analyzers of code on the submission. We also needed to make some changes within the DOMjudge PHP code and the HTML templates.

This process concluded by giving relevant feedback to the students about their code, and aid instructors on the assessment of assignments (it seems important to remark that we give the instructor 7 different types of modes in which the online judge will run or not the static and dynamic analyzers). Our developments provide an example to other teaching institutions in order to implement feedback based on code analysis when needed on an online judge. With the extension made on DOM-

judge, we detect and communicate most errors regarding memory and undefined behavior, and through linting and static analysis we analyze other properties of the code that are not detected by ASan or UBSan.

## 6.2. Future work

During the development of this project, we have come up with ideas and features that we would have been delighted to implement, but exceeded the focus of the project or could not include due to time restrictions.

Firstly, we would have loved to make the features available to other languages outside of C++. The main reason behind not doing this was that C++ is the main language used in our laboratories for the different subjects, and adding support to unused languages was not a priority. To do this, the judge should be able to identify the language from within the `run` script to know what static analyzers and instrumentation to run.

Another feature that caught our interest was including the information of whether past submissions had been accepted or rejected by the judge and the reason for it. This would have allowed us to develop more reactive messages and perform a more complex message generation. In the end, we did not include this feature due to time restrictions.

As it was discussed in the static analyzers conclusions, the analysis done on students submissions doesn't differ from problem to problem, running an analysis specific to the problem and providing the maintainer of the online judge an interface to do this easily, would solve this limitation. In [3], the authors propose the CAC++ library, construct tailored static analysis programs for C/C++ assignments. By integrating it, we could be able to detect the targeted logic errors.

We also thought about implementing the run options inside the test cases themselves. While functional, it is not the most user-friendly approach. An ideal solution would involve adding options and buttons in the user interface to select and activate these features. It would require work that falls outside the scope of this project, but would be positive to have.

Lastly, we would like to mention that our tutor's and other professors from our university have planned to use the DOMjudge integration developed in this thesis on some subjects in the computer science school at UCM for next year. Indeed, a teaching innovation project (number 71 of the 2024/25 call) has been awarded to a group of people including us. To improve this project it would be useful to have even more student submissions to see in more detail the most common errors among students. It will be useful to also consider different kinds of problems and subjects (for example, we coded in purpose a division by zero to see if a check was raised, but we did not find it raised in any of the +4000 submissions available)

With this thesis we have created a path to implement many other improvements for the feedback given by DOMjudge. This project is scalable and modular, so we expect that in the close future we can see this thesis as the basis for a better automated code assessment at the Universidad Complutense de Madrid.

# Chapter 7

## Personal Contributions

In this section, we will specify the job performed by each of the team members inside this project.

### 7.1. Rodrigo Burgos Sosa

During the initial phase of the project, I explored some of the different static analyzers that existed and the ones that could be more beneficial for our purpose. When looking at what each of them had to offer and decided to go for Clang and Cppcheck, I read through the different documentations provided and made sure to understand how they worked and the installation process. SonarQube and CodeChecker were other of the options that were considered, the second one for its Cross Translation Unit feature that we ended up deciding not to include. Many of the arguments added to the command lines of the static analyzers were chosen thanks to this process.

Through my investigation, I also realized that Clang-Tidy offered more value to us than Clang by itself, as it ran the same checks that the latter, in addition to some of its own. It, too, was simpler to set up than Clang, and we decided to use Clang-Tidy for these reasons.

To perform the initial execution of the analyzers with the student submissions, I developed the scripts to run them on all the submissions and obtain the parseable files in collaboration with Luis and Juan.

From this point onwards, I dedicated my time to the judge modifications. Firstly, I read through their documentation to understand how DOMjudge worked and the available features. I will use this space to express my gratitude personally to all the DOMjudge developers for maintaining such a complete documentation. Speaking of which, I was in charge of the communication with the developers, as the information they provided affected directly my part of the work. To do so, I subscribed to the development mailing list and expressed my doubts through there. The process is very simple, one has to subscribe (done to counter spam) and once the process is completed, I could send messages to the mailing list. Communication with the team was key in order to overcome some difficulties for our implementation. Jaap Eldering and Michael Vasseur, two of the main developers for the judge, answered everything we asked and guided some of our thoughts. I also explored everything

that the project had to offer, which is how we found out about all the repositories specially aimed at other developers attempting to make their contribution to the judge.

Thanks to these repositories, I was able to build the new images that were used during the implementation process. Despite how well the project is maintained, I still ran into some issues when using this resources to set everything up [42, 39]. For instance, I had to make sure that I enabled plugins to run when executing compose as root<sup>1</sup>, since at a given point, the option for plugins to run by default in Compose was disabled<sup>2</sup>. The **Symfony** plugin was affected by this, for instance. Also, some changes had to be made to some scripts to add features or options that are not the default behavior, as mentioned elsewhere in this thesis. A very important fix that had to be made was the fact that the web interface formatting at a certain point was not formatting properly and the styling was not loading, not only making the web page visually worse, but also making the whole user experience far less enjoyable and even removing some interactions, like the ability of adding problems to a contest, since certain buttons did not load. A message was sent to the team and this was a known issue that some users had reported already and are working on. They did suggest a possible fix, which should have been part of the process from the beginning anyway, which was running the Make dist on the code. This fortunately solved a problem that could have otherwise greatly detracted from the final product. All of this required a process of investigation and troubleshooting to find the issues and the solutions; many of them were found in GitHub and Stack Overflow threads and discussions, and some others by diving into specific documentations. At the end, more time was spent troubleshooting and fixing issues that I could have anticipated.

Once the images were created correctly, I developed the compose files that were used later by the team to test out the final changes. But, honestly, it was a feature that was mostly used by myself, so more than anything, it was a quality of life improvement. Either way, it allows anyone to easily deploy our version of DOMjudge with ease on their machines.

In order to use Docker on my machine, I downloaded Windows Subsystem for Linux 2 (WSL2)<sup>3</sup> since I personally use Windows on it. This allowed the whole setup to be easier and more convenient to use. It required a bit of setting up in order to make it work in synchronization with Docker.

By far, the part that consumed most of my time was understanding how to make the appropriate changes inside the judge. Most of my findings have already been exposed throughout this thesis, but many of the difficulties have not been explained. One of the most important milestones was the process of retrieving the number of submissions for a given problem by a student in order to use it to generate the feedback message. The process involved a lot of work to understand and make correctly. Firstly, I had to research how the API worked and where I could

---

<sup>1</sup><https://github.com/api-platform/api-platform/issues/2610>

<sup>2</sup><https://github.com/composer/composer/issues/11839> [1]

<sup>3</sup>Windows Subsystem for Linux (WSL) is a feature of Windows that allows to run a Linux environment on your Windows machine, without the need for a separate virtual machine or dual booting <https://learn.microsoft.com/en-us/windows/wsl/about> <https://learn.microsoft.com/en-us/windows/wsl/tutorials/wsl-containers> [24].



find the required information, to then identify the appropriate place from where the information should be requested, landing luckily inside the `judgedaemon.main.php`. Thanks to the DOMjudge developers, we understood that modifying the JudgeTask entity would be very beneficial, but the change was not as straight forwards as initially thought. A database migration was, of course, required to apply the change, and this required to understand at a surface level how to deal with Symfony and databases. Even with this, the migration process still ran into issues because I had to specify the `DATABASE_URL` environment variable inside the `.env.local` file. Many thanks to the threads that helped me solve the problems throughout this process.

Finding the most appropriate method to run the static analyzers and instrumentation was not an easy task either, and multiple options and configurations were tried in order to land on the final version. The process was in constant evolution, as a new method would lead the way into attempting to perform the same action but in a simpler or better implemented manner. Figuring out what specific changes had on the evaluation process was certainly doubt-inducing, and their effect would dictate the decisions taken. At some point, compilation of the code could be directly performed with instrumentation in the compile script, which meant that code had to be compiled only once per test case (if necessary), but DOMjudge did not tolerate working with two different compilation results if a test case performed compilation with instrumentation and another one did not, so we had to scrap the idea. In the end, the solution found was a far less complicated workaround.

Parts for the solution also involved modifying PHP and HTML code, which I was not familiar with before beginning this project. Many of the changes were made by observing sections of the code and trying to replicate them to achieve my goals.

Finally, I was in charge of writing in this thesis everything that had to do with the modifications made to the judge, apart of the common sections to the whole team (introduction, conclusions, ...), which were written by the three of us.

## 7.2. Juan Trillo Carreras

During the first phase of the project, I coordinated the rest of the participant's tasks, while Rodrigo explored superficially some of the different static analyzers and Luis the sanitizers, I studied thoroughly Cppcheck as it was the most complete and the one I felt was going to be more helpful for the feedback to the students when improving the answers of DOMjudge. After our first exploration, I started coding together with Rodrigo and Luis the initial bash to get different submissions to be examined by the different analyzers chosen. I made the final modifications of the script. After completing the script in `bash` we decided to make different Python scripts to examine the "raw" output of the analyzers. Through this second phase we had various changes in how to read the output for the tools as they had different options for the outputs. For example, at first I wrote one script for taking the SARIF output of Clang, but at the end we decided to take the YAML output. All the Python scripts were going to be ran by the initial bash script, so I had to make more changes to the initial script as I progressed with the other scripts. I was in charge of the main script used for Clang and Cppcheck. At first, we united all the

errors on a general CSV for all the tools. However, we ended up changing it for three different JSONs, one for Clang-Tidy, other for Cppcheck and one last for the GCC outputs.

On the study of the static analyzers used in the project, I studied GCC with the corresponding analysis of the checks that the static analyzer provided and determined that they were not clear enough or sufficient to communicate to the student. I also studied Cppcheck and had some troubles finding an official documentation for the tool, however I managed to make an exhaustive study on all the checks raised in the student submissions that our tutor provided us. I made several reports on each of the errors with help of a Python script and developed for parsing the XML output of the tool. I have also made a summary with graphs for the Cppcheck checks.

For storing everything and dividing the tasks, I created a workspace on Google Drive to work together with Luis on the study of the different static analyzers and sanitizers.

During the study of the sanitizers I devoted my time learning regular expressions as I had never used them before. I had various struggles to clean the UBSan output. I could see a pattern between some errors that were specified as the same but were not counted as one type of error. After cleaning the data extracted with my own developed Python script, I made an analysis of the data and determined the most relevant checks the sanitizer raised.

On the final phase of the project, I developed the script that shows the output of the Cppcheck checks to the student depending on the number of tries the student had on the assignment as well as the UBSan one (to see the scripts mentioned till this part please check the GitHub repository <sup>4</sup>).

During the writing of the memory of this thesis I was in charge of writing the abstract and the keywords (both in Spanish and in English).

On the introduction chapter I polished the goals and divided them in different sections. I added the organization of the thesis (including all chapters and short descriptions of them).

On the state of the art, I wrote the description, the online judges section, where I explained what they were and a short example on where are they used, and their basic way of working, I then introduced DOMjudge, explaining what is it and where is actually used: Next I together with Luis wrote the list of different possible results. I also wrote the basic functionalities of DOMjudge. I was also in charge of writing what is static analysis. I also explained what is Cppcheck, GCC, and UBSan as I was the one who studied them.

On Chapter 3, I wrote the short description of the chapter as well as the process followed for all the static analyzers, as well as the in-depth study of all the relevant checks raised in Cppcheck and UBSan, the summary of all checks raised by the static analyzers together with the conclusion of the chapter.

For the subsequent chapters up to and including Chapter 8, I just helped Rodrigo with overviewing the process. On chapter 8 I provided together with Luis the final and polished parses that can be found on the repository. I also provided the actual test case scenario code to check if everything was going accord to our expectations.

---

<sup>4</sup><https://github.com/Jantri-3/analysis-of-DOMjudge-submissions>

On Chapter 6, I wrote all the discussions and conclusions of this thesis with a part of the future work that could be developed. I also wrote Appendix A.2, Appendix A.3 and Appendix A.4.

Building the bibliography following our tutor's advises was also one of my responsibilities, I made the template of the most used referencing method in this thesis `@misc`, I had to seek solutions to how to link the actual web pages and having a coherent way of referencing them.

I coordinated the team by dividing the parts of this thesis, but we all overlooked the process of all team members.

I modified the plain text version of this memory to `LATEX` with help of Rodrigo as we are new to this "coding language". We had to learn how to add new packages and use them during that process. In this process, I also created the color palette seen in this project (used mainly in the tables and in the in-depth analysis of Cppcheck and UBSan)

## 7.3. Luis Esteban Velasco

During the research phase I went in depth over all the Clang checks and wrote a preliminar list with a short description and if they were useful for students using DOMjudge or not. Then, when Rodrigo noticed how much more useful was Clang-Tidy I also went over it. I also studied how ASan worked and the different errors it can detect. When we still had the bash script, I wrote most of the Python code for the scripts that were supposed to organize the "raw" output, sometimes alone and sometimes helping Juan, as I am more familiar with Python.

We decided to translate the bash to Python and then we pulled apart the parsing of the output from the running of the different tools and after that I made the scripts that parsed the output into a JSON file. Changing the bash to Python was done by Rodrigo and me, pair programming. We had to fix a few problems in the implementation too, which were spotted by our tutor. Those problems ultimately resulted in the decision to separate the scripts that run the tools and organize the output from the ones that analyze it.

I also wrote the code that generated most of the results in the analysis part and the script that generated the graphs. I did that with the help of Jupyter notebook, as it has better integration with matplotlib.

Both the parsing and analysis scripts were just a preliminary version, Juan had to make some changes on the analysis scripts for Cppcheck, GCC, and UBSan, and polish them for better readability, I helped him with some of his doubts about the code, as I tend to use more complex Python.

After that, our tutor had a great idea, to parse the information of the submissions ordered by problem, and then by group. This made me change the parsing scripts for ASan and Clang-Tidy completely. This also made possible to analyze the transitions to AC, which was actually why our tutor wanted to do it. With some of his help, I wrote the code for it and also changed many of the older analysis scripts, mainly for extracting some wider statistics on global results of the submissions, and on the submissions and test cases that had ASan errors.

One of the major difficulties I had was writing the code to analyze the transitions to AC within ASan. At first, I didn't even know how to check for transitions, it was only with my tutor's help that I managed to parse the output correctly, and then I wrote the code to analyze transitions to AC for the static analyzer, the main difference between ASan and Clang-Tidy was that the transitions in ASan needed to be analyzed from test case to test case. This, at first, would seem easy but due to how ASan was parsed and some errors I made while writing it, made it a nightmare. Before reaching the correct solution I reached a result that looked right. Thanks to reviewing the results, I later noticed the mistakes and, using the great debugger PyCharm offers, I managed to detect those errors and fix the code.

I also wrote the final parser for Clang-Tidy, which is supposed to run within DOMjudge, and ASan. In the Clang-Tidy parser instead of writing several `if` statements, I used a dictionary that takes the diagnostic name and returns as its value a lambda function that takes two arguments (number of tries and the error information) and passes them to the function that ultimately generates the message. This way, instead of running a bunch of string comparisons, it just uses the hash function and returns the lambda. In the ASan parser I experimented with parsing the stack trace and possible messages. Unfortunately, I ultimately decided it was too hard to write useful explanatory messages as multiple things could have triggered the same error. It was also too time-consuming to write a good stack trace parser and then test it to make sure it works correctly. Doing all of this work wouldn't bring more value to the feedback, as just outputting the entire ASan message would be better, maybe you could exclude the specific addresses, but that is it. Later, we decided to add messages to it, and when I was writing them I decided to just talk about some possible causes, instead of the most probable one.

Regarding the memory, I wrote most of the motivation, in the state of the art, I wrote the section about Instrumentation and its part about ASan. I also wrote about the Clang Static analyzer. In the static analysis subsection I wrote the introduction, the sections about DOMjudge results and Clang, and the written conclusions. In the sanitizers subsection I wrote the ASan section, the introduction and conclusions. I also helped with other parts by providing feedback and some proof-reading to find grammar mistakes and typos, and adding some paragraphs to other sections.

I also read multiple papers during the research phase and during the writing of the thesis, specially the paper on [3], to investigate if it could be applied as future work and bring value to our implementation. I also read [22] and [40] to better understand previous work about online judges. The paper on better feedback for online judges had a very similar motivation to ours but a very different solution, which made me think it could be added to our implementation, but I saw the need of having many test cases as a disadvantage that would make instructors work more, an issue which our implementation tries to solve.

While I did not help program or set up anything related to the DOMjudge, I had to set up WSL (Ubuntu 22.04.3) and Docker Desktop in my system to test for a specific error we were having with the DOMjudge web interface and JQuery. While running the containers was fairly easy as Rodrigo had done a good job when making the `compose.sh` that ran the Docker Compose and some instructions, I had a hard time setting up WSL, due to some errors in the WSL I had installed before this

---

(which I had to uninstall), connecting Docker Desktop to WSL (it was connected to another version of it). I also had a hard time running the script with bash until I realized it was due to the format of newlines (which was easily fixed with the `sed` command).



# Bibliography

- [1] Jordi Boggiano et al. Composer: dependency manager for PHP, 2012. URL: <https://github.com/composer>.
- [2] Guanzhong Chen, Tudor Brindus, et al. DMOJ: Modern online judge, 2011. URL: <https://dmoj.ca>.
- [3] Pedro Delgado-Pérez and Inmaculada Medina-Bulo. Customizable and scalable automated assessment of C/C++ programming assignments. *Computer Applications in Engineering Education*, 28(6):1449–1466, 2020.
- [4] Valgrind developers. Valgrind, 2002. URL: <https://valgrind.org/>.
- [5] CMS development team. CMS contest management system, 2012. URL: <https://cms-dev.github.io>.
- [6] Directi. Codechef, 2009. URL: <https://www.codechef.com>.
- [7] Jaap Eldering, Thijs Kinkhorst, et al. DOMjudge, 2004. URL: <https://www.domjudge.org/>.
- [8] Jaap Eldering, Thijs Kinkhorst, et al. DOMjudge API documentation, 2004. URL: <https://www.DOMjudge.org/demoweb/api/doc>.
- [9] Jaap Eldering, Thijs Kinkhorst, et al. DOMjudge Github repository, 2004. URL: <https://github.com/DOMjudge>.
- [10] Free Software Foundation. Dist (automake): what goes in a distribution?, 2006. URL: [https://www.gnu.org/software/automake/manual/1.10/html\\_node/Dist.html](https://www.gnu.org/software/automake/manual/1.10/html_node/Dist.html).
- [11] Free Software Foundation. GNU GRUB manual 2.12n, 2024. URL: <https://www.gnu.org/software/grub/manual/>.
- [12] ICPC Foundation. International Collegiate Programming Contest, 1970. URL: <https://icpc.global>.
- [13] Mario Fernández González. *Evaluación y despliegue de herramientas de gestión de concursos de programación*. Bachelor’s thesis, Universidad de Cantabria, 2022. URL: <https://hdl.handle.net/10902/25832>.
- [14] LLVM Developer Group. Clang static analyzer, version 17.0.6, 2009. URL: <https://clang-analyzer.llvm.org> (visited on 2024).

- [15] Anthony Heddings. What is chroot on Linux and how do you use it?, 2020. URL: <https://www.howtogeek.com/devops/what-is-chroot-on-linux-and-how-do-you-use-it/>.
- [16] Colin Hughes. Project Euler, 2001. URL: <https://projecteuler.net>.
- [17] Docker Inc. Docker Compose overview | Docker docs, 2013. URL: <https://docs.docker.com/compose/>.
- [18] Docker Inc. Docker docs, 2013. URL: <https://docs.docker.com/>.
- [19] Linux kernel development community. Control groups version 1, The Linux Kernel documentation, 2024. URL: <https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/index.html>.
- [20] Rüdiger Landmann, Jack Reed, et al. *The GRUB loader*. In *Red Hat Enterprise Linux 5: Installation Guide*. 2024. Chapter 9. URL: [https://access.redhat.com/documentation/en-us/red\\_hat\\_enterprise\\_linux/5/html/installation\\_guide/ch-grub](https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/5/html/installation_guide/ch-grub).
- [21] Kaibo Liu, Yudong Han, Jie M. Zhang, Zhenpeng Chen, Federica Sarro, Mark Harman, Gang Huang, and Yun Ma. Who judges the judge: an empirical study on online judge tests. In René Just and Gordon Fraser, editors, *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, pages 334–346. ACM, 2023.
- [22] Anaga Mani, Divya Venkataramani, Jordi Petit Silvestre, and Salvador Roura Ferret. Better feedback for educational online judges. In *Proceedings of the 6th International Conference on Computer Supported Education, Volume 2: Barcelona, Spain, 1-3 April, 2014*, pages 176–183. SciTePress, 2014.
- [23] Daniel Marjamäki. Cppcheck, 2007. URL: <https://sourceforge.net/p/cppcheck/wiki/ListOfChecks/>.
- [24] Microsoft. WSL documentation, 2016. URL: <https://learn.microsoft.com/en-us/windows/wsl/>.
- [25] IOI organization. International Olympiad in Informatics, 1989. URL: <https://ioinformatics.org>.
- [26] Sachin Patil. What is a makefile and how does it work?, 2018. URL: <https://opensource.com/article/18/8/what-how-makefile>.
- [27] Jordi Petit and Salvador Roura. Jutge.org, 2006. URL: <https://jutge.org>.
- [28] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.
- [29] Symfony SAS. Symfony. High performance PHP framework for web development, 2005. URL: <https://symfony.com/>.
- [30] Carlos Solar Sastre. *Desarrollo de una Plataforma Orientada al Refuerzo en la Evaluación del Nivel de Programación*. Bachelor’s thesis, Universidad Politécnica de Madrid, 2020. URL: <https://oa.upm.es/63124/>.



- [31] UNICOM Systems. Purify, 1992. URL: <https://www.unicomsi.com/products/purifyplus/>.
- [32] UNICOM systems. Quantify, 1992. URL: <https://www.unicomsi.com/products/quantify/>.
- [33] Clang Team. AddressSanitizer — Clang 19.0.0git documentation, 2012. URL: <https://clang.llvm.org/docs/AddressSanitizer.html> (visited on 2024).
- [34] Clang Team. Clang-Tidy checks, 2024. URL: <https://clang.llvm.org/extra/clang-tidy/checks/list.html>.
- [35] Clang Team. UndefinedBehaviorSanitizer — Clang 19.0.0git documentation, 2012. URL: <https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html#ubsan-checks> (visited on 2024).
- [36] DOMjudge team. Domjudge manual, 2004. URL: <https://www.DOMjudge.org/docs/manual/8.2/index.html>. Manual for the DOMjudge.
- [37] ICPC team. Contest API, 2010. URL: [https://ccs-specs.icpc.io/draft/contest\\_api#introduction](https://ccs-specs.icpc.io/draft/contest_api#introduction). Contest API documentation.
- [38] Tpoint Tech. Linux arguments, 2011. URL: <https://www.javatpoint.com/linux-arguments>.
- [39] user3001829. How to copy files from host to Docker container?, 2014. URL: <https://stackoverflow.com/questions/22907231/how-to-copy-files-from-host-to-docker-container>.
- [40] Szymon Wasik, Maciej Antczak, Jan Badura, Artur Laskowski, and Tomasz Sternal. A survey on online judge systems and their applications. *ACM Comput. Surv.*, 51(1):3:1–3:34, 2018. eprint: <https://arxiv.org/abs/1710.05913>.
- [41] Alex Williams. What is Symfony?, 2022. URL: <https://builtin.com/software-engineering-perspectives/symfony>.
- [42] Catch Zeng. Tool - Docker build not showing any output from commands (Dockerfile RUN), 2016. URL: <https://makeoptim.com/en/tool/docker-build-not-output/>.



## Offline analysis details

### A.1. Most relevant Clang-Tidy diagnostics

In Table A.1 we show the number of occurrences of each diagnostic within all of the submissions, shown in the second column. But, in order to avoid any kinds of bias related to diagnostics appearing multiple times within a team or a problem we added two other columns. The third column counts the number of team’s submissions for a problem which contained the diagnostic, and the fourth column contains the number of problems which have submissions containing the diagnostic. The last column displays the number of transitions from not accepted submissions to accepted ones when a diagnostic is fixed. The table is ordered by transitions to AC then by total occurrences. It contains the first 40 after removing checks concerning a specific project, security and concurrency, some specific checks that are not very relevant are also removed, they are related with very opinionated style requirements that students do not follow.

Table A.1:

Name	Submission	Team	Problem	AC Trans
performance-avoid-endl	564	263	48	18
misc-no-recursion	407	196	25	11
readability-implicit-bool-conversion	156	69	19	8
bugprone-exception-escape	702	322	26	7
bugprone-narrowing-conversions	481	241	38	7
cppcoreguidelines-pro-type-member-init	277	134	29	7
google-explicit-constructor	204	118	15	7
bugprone-integer-division	21	11	1	6
modernize-use-nodiscard	314	159	21	5

Continued on next page

Table A.1: (Continued)

Name	Submission	Team	Problem	AC Trans
readability-make-member-function-const	134	85	14	5
cppcoreguidelines-prefer-member-initializer	44	30	8	5
modernize-loop-convert	184	94	25	4
modernize-use-transparent-functors	123	76	15	4
readability-simplify-boolean-expr	62	40	15	4
readability-function-cognitive-complexity	52	16	9	4
bugprone-easily-swappable-parameters	285	172	34	3
google-runtime-int	218	87	24	3
performance-for-range-copy	117	64	11	3
cppcoreguidelines-avoid-const-or-ref-data-members	111	70	11	3
clang-diagnostic-empty-body	62	27	20	3
clang-analyzer-deadcode.DeadStores	57	32	15	3
bugprone-branch-clone	56	25	12	3
modernize-return-braced-init-list	25	9	4	3
cppcoreguidelines-use-default-member-init	255	148	17	2
bugprone-reserved-identifier	128	64	6	2
misc-non-private-member-variables-in-classes	107	52	15	2
performance-unnecessary-value-param	78	56	21	2
modernize-use-using	67	26	12	2
hicpp-use-emplace	49	32	16	2
misc-unused-parameters	30	20	16	2
clang-diagnostic-parentheses-equality	5	2	1	2
clang-diagnostic-error	91	57	18	1
clang-diagnostic-return-type	60	41	8	1
cppcoreguidelines-non-private-member-variables-in-classes	48	34	10	1
bugprone-suspicious-semicolon	38	16	11	1
cppcoreguidelines-macro-usage	36	17	11	1

Continued on next page

Table A.1: (Continued)

Name	Submission	Team	Problem	AC Trans
readability-misleading-indentation	33	16	13	1
modernize-pass-by-value	24	16	9	1
modernize-macro-to-enum	21	11	11	1

## A.2. Cppcheck error list

Here is the list of the most relevant checks of Cppcheck found and the reasons of why we chose them:

- `missingReturn` (error)
  - From 46 occurrences **WA**: 6.52%, **CE**: 17.39%, **RTE**: 4.35%, **TLE**: 2.17%, **AC**: 69.57%. As the DOMjudge does only rely on output to state an AC, we can see how this result is happening. However this is a very poor coding practice and could cause problems.
  - "Found an exit path from function with non-void return type that has missing return statement"
  - If not handled will probably cause errors.
  - Check should be communicated although it may not cause any problems, it aids the student in a didactic way.
- `uninitStructMember` (error)
  - From 28 occurrences **WA**: 57.14%, **CE**: 21.43%, **RTE**: 10.71%, **AC**: 10.71%. Probably an unintended error.
  - "Uninitialized struct member: \*\*\*\*"
  - Part of the struct is not given type / initialized by the programmer but it was defined by themselves this is quite suspicious and should be communicated.
- `unreadVariable` (style)
  - From 300 occurrences **WA**: 36.12%, **CE**: 7.02%, **RTE**: 11.71%, **TLE**: 13.04%, **AC**: 32.11%.
  - "Variable '\*\*' is assigned a value that is never used."
  - Not reaching a variable (which means that a variable was not read during execution) is the consequence of: part of the code is unreachable, there is an infinite loop (TimeLimitException), the program cannot be compiled completely (CompileError) or a mix of those reasons.
  - Should be communicated in case of TLE, CE and WA.
- `shadowFunction/shadowVariable/shadowArgument` (style)

- Except from shadowFunction that has just a 13.33% of AC the rest have a relatively high acceptance rate (around 40%) in comparison.
  - "Local variable '\*\*' shadows outer function/Variable/Argument"
  - The programmer may be inadvertently using a name already used by the program thinking that they are required to be connected when they are not.
  - Should be communicated in case of not being AC.
- knownConditionTrueFalse (style)
    - From 52 occurrences WA: 28.85%, CE: 9.62%, RTE: 23.08%, TLE: 32.69%, AC: 5.77%.
    - "Condition 'arbol.empty()' is always false"
    - Almost never arises when the program is AC. Although this might be a conscious decision of the student, this can lead to various problems.
    - Should be communicated for maybe aiding the student in a didactic way.
  - redundantCondition (style)
    - From just 4: 3 are RTE, and just 1 is TLE.
    - "Redundant condition:  
j==0. 'j==0 || (j==0 AND i==-1)' is equivalent to 'j==0'"
    - Tightly related to RTE and TLE (caused mainly by unconscious infinite loops).
  - multiCondition (style)
    - 2 occurrences both of them RTE.
    - "Expression is always true because 'else if' condition is opposite to previous condition at line 56."
    - Tightly related to RTE.
  - duplicateExpression (style)
    - From 17 occurrences WA: 29.41%, CE: 11.76%, RTE: 5.88%, TLE: 52.94%, AC: 0%
    - "Same expression on both sides of '>'."
    - Seems highly related to TLE (Infinite loops with while conditions) .
  - duplicateBreak (style)
    - As one could imagine this check happens very unfrequently in our case we have just seen one case which was actually accepted.
    - "Consecutive return, break, continue, goto or throw statements are unnecessary."

- This is a bad coding practice.
  - Check should be communicated although it may not cause any problems, it aids the student in a didactic way.
- `noConstructor` (style)
    - With just 2 occurrences, one was TLE and the other AC.
    - `"The class 'Supermercado' does not declare a constructor although it has private member variables which likely require initialization."`
    - Could cause problems (in our dataset it lead to errors).
    - Should be communicated in any case different from AC.
  - `unreachableCode` (style)
    - Surprisingly this just happens once, and its AC, after further investigation, we noticed that this was because it was related to the *duplicateBreak* error.
    - `"Statements following 'return' will never be executed."`
    - Could cause problems as it can be highly associated with infinite loops or conditions that will never actually be met because of the code.
    - Should be communicated in any case different from AC.
  - `knownEmptyContainer` (style)
    - Just one occurrence, an OLE verdict, as the example shows, it was outputting an empty result with probably a fixed output format, and as it was iterating over it, it made it exceed the output limit.
    - `"Iterating over container 'sol' that is always empty."`
    - Will lead to problems, highly related to OLE.
    - Should be communicated in any case different from AC.
  - `clarifyCondition` (style)
    - Just one occurrence in the dataset, surprisingly less than we originally theorized however it is relevant and the verdict was WA.
    - `"Boolean result is used in bitwise operation. Clarify expression with parentheses."`
    - Will cause problems, highly related to WA.
    - Should be communicated in any case different from AC.
  - `uninitvar` (warning)
    - From 14 occurrences `WA`: 28.57%, `RTE`: 28.57%, `TLE`: 19.05%, `AC`: 23.81%

- "Uninitialized variable: `***.h`"
- Not initialized variable (it is a warning because as you can see is mainly associated with header files wrongly interpreted).
- `selfAssignment` (warning)
  - Just 9 occurrences all of them WA, as expected, this is not a common error but its almost always "fatal" for the correctness of the answer.
  - "Redundant assignment of `'maneras[y]'` to itself."
  - Highly associated with WA.
- `negativeContainerIndex` (warning)
  - Just 6 occurrences, 2 of them caused an RTE which is normal as if it actually gives a negative cointainer index it will result in a run-time exception.
  - "Either the condition `'adyacentes[i]! = anterior'` is redundant, otherwise there is negative array index -1."
  - Highly associated with RTE, might cause runtime exceptions.
- `containerOutOfBounds` (warning)
  - WA: 4.76%, CE: 23.812%, RTE: 66.67%, AC: 4.76%. These results makes sense, remember we are treating with static analysis so this error arises when a variable tries to access a part of a variable which can be empty or out of bounds, depending on the actual value of both variables, this can lead to different accesses resulting majorly in RTE.
  - "Either the condition `'****[0].empty()'` is redundant or expression `'****[0].front()'` cause access out of bounds."
  - Highly associated with RTE, might cause runtime exceptions.
- `identicalInnerCondition` (warning)
  - Just one occurrence, derived in an AC verdict, its definitely not an intended error so it must be notified.
  - "Identical inner `'if'` condition is always true."
  - Probably not intended.
  - Check should be communicated although it may not cause any problems, it aids the student.
- `constStatement` (warning)
  - From 7 occurrences WA: 14.29%, CE: 28.57%, RTE: 28.57%, TLE: 14.29%, AC: 14.29%. It depends on the case.
  - Case 1



- "Redundant code: Found a statement that begins with bool constant."
  - Highly associated with WA
  - Should be communicated as it can cause problems.
- Case 2
  - "Redundant code: Found unused array access."
  - Unused variable
  - Not relevant for feedback.
- Before going any further, now we are going to analyze **performance** errors, these imply always a bad/poor coding practice which means that they might not imply a judge verdict, however although not useful for the student to correct their answer, this kind of errors are very useful for them to integrate good coding practices to their codes.
- `passedByValue` (**performance**)
  - From 94 occurrences **WA**: 17.58%, **CE**: 3.3%, **RTE**: 17.78%, **TLE**: 28.57%, **AC**: 30.77%. Just a poor coding practice, could deliver any kind of verdict, but it not the main reason for any of them.
  - "Function parameter 'tallaNecesitamos' should be passed by const reference."
  - Poor coding practice.
  - Check should be communicated although it may not cause any problems, it aids the student in a didactic way.
- `stlFindInsert` (**performance**)
  - From 22 occurrences **WA**: 10%, **CE**: 5%, **RTE**: 5%, **TLE**: 5%, **AC**: 75%. Poor coding practice, does not imply any verdict. However it could be corrected.
  - "Searching before insertion is not necessary. Instead of 'personas[]=indPersona' consider using .tryemplace
  - Poor coding practice.
  - Check should be communicated although it may not cause any problems, it aids the student in a didactic way.
- `iterateByValue` (**performance**)
  - From 8 occurrences **WA**: 14.29%, **CE**: 28.57%, **TLE**: 28.57%, **AC**: 28.57%. Poor coding practice, does not imply any verdict. However it could be corrected.
  - "Range variable 'cancion' should be declared as const reference."

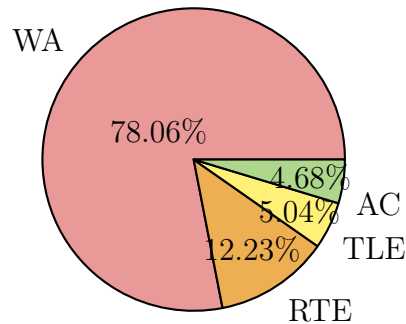
- Poor coding practice.
- Check should be communicated although it may not cause any problems, it aids the student in a didactic way.
- `useInitializationList` (performance)
  - From 4 occurrences 3 are TLE and just one is CE. Poor coding practice, does not imply any verdict. However it could be corrected.
  - "Variable 'cola' is assigned in constructor body. Consider performing initialization in initialization list."
  - Poor coding practice.
  - Check should be communicated although it may not cause any problems, it aids the student in a didactic way.

### A.3. UBSan error list

Here is a list of the rest of the judge results on the remaining errors that UBSan detects:

- Error: signed integer overflow: 278.
  - Usually wrong answer, communicate in case of not AC.

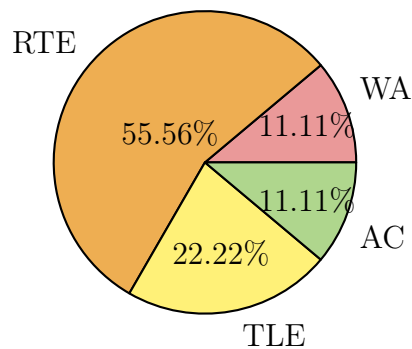
Figure A.1: Verdicts for *signed integer overflow*



- Error: reference binding to null pointer: 52.
  - Really important check, should be communicated.
    - WA: 1.92%; RTE: 98.08%
- Error: member access within null pointer: 23.
  - Communicate in case of not AC (probably not intended).
    - WA: 4.35%; RTE: 95.65%
- Error: applying non-zero offset to null pointer: 10.

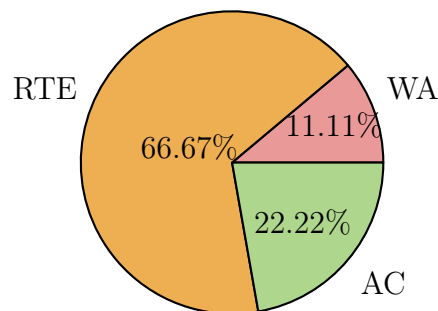
- Communicate in case of not AC (probably not intended).
  - RTE: 80%; TLE: 20%
- Error: execution reached the end of a value-returning function without returning a value: 36.
  - Poor programming practice, should throw an error, communicate the feedback (the function should be of type void).
    - Figure A.2

Figure A.2: Graph of verdicts for error: *execution reached the end of a value-returning function without returning a value*



- Error: load of value , which is not a valid value for type 'bool': 12.
  - Poor programming practice, a bool type should be only referenced via True / False not integers communicate.
    - WA: 66.67%; AC: 33.33%
- Error: pointer index expression with base: 8.
  - Communicate in case of not AC (probably not intended) - the base refers to a position in hex 0x... which is usually not intended for users to use.
    - Figure A.3

Figure A.3: Graph of verdicts for error: *pointer index expression with base*



## A.4. Table of code analysis

On this table we provide an explanation as to why checks are raised when present on an student code and, if possible, when to communicate them to the student. The last column says when should we communicate that error depending on the response of the judge to said code for example !AC means only communicate when the response of the judge is different from AC; TLE is to communicate the error when the judge response is TLE and Yes is to communicate the error to the student regardless of the judge's response. The main purpose of this table is to find overlapping checks between Cppcheck and Clang-Tidy static analyzers.

Legend of the table:

- Unique check/error
- Repeated check/error in both tools
- chosen check/error

Error ID	Explanation	Communication
uninitvar	variable is not initialized by the programmer but it was defined by themselves	Yes
unreadVariable	Not reaching a variable could be caused by various reasons: part of the code is unreachable, there is an infinite loop(TLE), the program cannot be compiled completely(CE) or a mix of those reasons	TLE, CE and WA
shadowFunction	Programmer may be inadvertently using a name already used by the program thinking that they are required to be connected when they are not.	!AC
shadowVariable	Programmer may be inadvertently using a name already used by the program, thinking that they are required to be connected when they are not	!AC
shadowArgument	The programmer may be inadvertently using a name already used by the program thinking that they are required to be connected when they are not.	!AC

Error ID	Explanation	Communication
knownConditionTrueFalse	Although this might be a conscious decision of the student, this can lead to various problems	Yes
redundantCondition	<code>j==0    (j==0 &amp;&amp; i==1)</code> is equivalent to <code>'j==0'</code>	No (=misc-redundant-expression)
multiCondition	Expression is always true	No (=misc-redundant-expression)
duplicateExpression	Duplicate expression at both sides of a comparison ( <code>&gt;</code> , <code>&lt;</code> , <code>==</code> )	No (=misc-redundant-expression)
duplicateBreak	Consecutive return, break, continue, goto or throw statements that are unnecessary.	Yes
uninitStructMember	Part of the struct is not given type / initialized by the programmer but it was defined by themselves.	Yes
noConstructor	Class does not declare a constructor, although it has private member variables which likely require initialization	!AC
unreachableCode	Statements following 'return' will never be executed.	!AC
knownEmptyContainer	Iterating over container that is always empty.	!AC
clarifyCondition	Boolean result is used in bitwise operation. Should clarify the expression with parentheses.	!AC
selfAssignment	Redundant assignment of <code>**[y]</code> to itself.	Yes
negativeContainerIndex	Either a condition is redundant, or there is negative array index - 1.	Yes
containerOutOfBounds	Either a condition is redundant, or there is an out of bounds access.	Yes
identicalInnerCondition	Identical inner 'if' condition is always true.	Yes
passedByValue	Function parameter Could be passed by const reference.	Yes

Error ID	Explanation	Communication
stlFindInsert	Instead of 'Array[Index]=Content' consider using 'Array.try_emplace(Index, Content);'	Yes
iterateByValue	Range variable should be declared as const reference.	Yes
useInitializationList	A variable is assigned in constructor body. Consider performing initialization in initialization list.	Yes
missingReturn	Found an exit path from function with non-void return type that has missing return statement.	Yes
legacyUninitvar	Uninitialized variable.	Yes
internalAstError	AST broken E.G: binary operator '=' doesn't have two operands.	Yes
selfInitialization	Member variable is initialized by itself.	!AC
constStatement	Redundant code (Not accessed arrays, constant statements starting with a non-variable bool).	No (=misc-redundant-expression)
bugprone-narrowing-conversions	Checks for silent narrowing conversions like <code>int i = 0; i += 0.1;</code>	Yes
bugprone-suspicious-semicolon	Ternary operator wrongly used	Yes
bugprone-integer-division	Division between ints assign to the result as a double, division integer then casted to double.	Yes
bugprone-branch-clone	Checks for branches in conditional statements that are identical.	Yes
misc-redundant-expression	Detect redundant expressions which are typically errors due to copy-paste.	Yes
performance-for-range-copy	Finds for ranges where the loop variable is copied in each iteration but it would suffice to obtain it by const reference.	Yes
performance-unnecessary-value-param	Finds expensive to copy types where a const reference would suffice.	Yes
performance-avoid-endl	Checks for usage of <code>std::endl</code>	Yes

Table A.2: Static analyzers error's explanations and communication

## Appendix B

# Instructions for Docker setup

This appendix contains a detailed walk-through with the commands that have to be executed and the steps that have to be followed in order to set up the working environment inside Docker. Before being able to follow these steps, one has to make sure that they have cgroup v1 enabled and Docker installed.

The first step is to initialize a database container for the domserver to store its data. In our case, we will follow DOMjudges's guide and initialize a MariaDB container:

```
docker run -it --name dj-mariadb -e MYSQL\_ROOT\_PASSWORD=rootpw  
→ -e MYSQL\_USER=domjudge -e MYSQL\_PASSWORD=djpw -e  
→ MYSQL\_DATABASE=domjudge -p 13306:3306 mariadb  
→ --max-connections=1000
```

This will set the root password to *rootpw*, create a MySQL user named *domjudge* with password *djpw* and create an empty database named *domjudge*. It will also expose the server on port 13306 on the local machine. These are the default parameters set by the DOMjudge team. With the database now created, we can create a DOMServer instance running the following command:

```
docker run --link dj-mariadb:mariadb -it -e MYSQL\_HOST=mariadb  
→ -e MYSQL\_USER=domjudge -e MYSQL\_DATABASE=domjudge -e  
→ MYSQL\_PASSWORD=djpw -e MYSQL\_ROOT\_PASSWORD=rootpw -p  
→ 12345:80 --name domserver domjudge/domserver:latest
```

Note that the tag "latest" indicates that the latest available version in Docker Hub must be used, which does not necessarily correspond to the most updated version published in the GitHub main branch. This will become relevant later down the line. The passwords for the admin account and judgehost – which is used to connect to the API – can be obtained at any time by accessing the corresponding files inside the domserver container. This judgehost password must be used to start the judgehost and link it to domserver. To run a single judgehost, we ran the following command:

```
docker run -it --privileged -v /sys/fs/cgroup:/sys/fs/cgroup:ro
↳ --name judgehost-0 --link domserver:domserver --hostname
↳ judgedaemon-0 -e DAEMON\_ID=0 -e
↳ JUDGEDAEMON\_PASSWORD=<judgehost password>
↳ domjudge/judgehost:latest
```

Where <judgehost password> must contain the value retrieved earlier. This can be run multiple times to create as many instances of judgehost as necessary. The name and hostname must be changed to allow for this for each instance.