Software Plagiarism Detection on Intermediate Representation

Bachelor's Thesis of

Niklas Rainer Heneka

At the KIT Department of Informatics
KASTEL – Institute of Information Security and Dependability

First examiner:	Prof. Dr. Ralf H. Reussner
Second examiner:	Prof. Dr.-Ing. Anne Koziolek
First advisor:	M.Sc. Timur Sağlam
Second advisor:	M.Sc. Larissa Schmid

26. June 2023 – 26. October 2023

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself. I have not used any other than the aids that I have mentioned. I have marked all parts of the thesis that I have included from referenced literature, either in their original wording or paraphrasing their contents. I have followed the by-laws to implement scientific integrity at KIT.

Karlsruhe, 26. October 2023

. . . .

(Niklas Rainer Heneka)

Abstract

Source code plagiarism is a widespread problem in computer science education. To counteract this, software plagiarism detectors can help identify plagiarized code. Most state-of-the-art plagiarism detectors are token-based. It is common to design and implement a new dedicated language module to support a new programming language. This process can be time-consuming, furthermore, it is unclear whether it is even necessary. In this thesis, we evaluate the necessity of dedicated language modules for Java and C/C++ and derive conclusions for designing new ones. To achieve this, we create a language module for the intermediate representation of LLVM. For the evaluation, we compare it to two existing dedicated language modules in JPlag. While our results show that dedicated language modules are better for plagiarism detection, language modules for intermediate representations show better resilience to obfuscation attacks.

Zusammenfassung

Plagiate von Quelltext sind ein weit verbreitetes Problem in der Informatik. Um dem entgegenzuwirken, können Software-Plagiatsdetektoren helfen, plagiierten Code zu erkennen. Die meisten modernen Plagiatsdetektoren sind token-basiert. Das Entwerfen und Implementieren eines neuen dedizierten Sprachmoduls zur Unterstützung einer neuen Programmiersprache ist dabei üblich. Dieser Prozess kann zeitaufwändig sein, und es ist unklar, ob er überhaupt notwendig ist. In dieser Arbeit evaluieren wir die Notwendigkeit von dedizierten Sprachmodulen für Java und C/C++ und leiten Schlussfolgerungen für das Design neuer Sprachmodule ab. Um dies zu erreichen, erstellen wir ein Sprachmodul für die Zwischensprache von LLVM. Für die Evaluation vergleichen wir es mit zwei bestehenden dedizierten Sprachmodulen in JPlag. Während unsere Ergebnisse zeigen, dass dedizierte Sprachmodule besser für die Erkennung von Plagiaten geeignet sind, zeigen Sprachmodule für Zwischensprachen eine bessere Widerstandsfähigkeit gegen Verschleierungsangriffe.

Contents

List of Figures

List of Tables

1. Introduction

Software plagiarism is a widespread issue in computer science education [7]. It is common for students to complete programming assignments. They are often assigned to get solved at home within several days or weeks [17, 18]. These tasks are a practical test of the student's skills [12] and can replace classic exams. Therefore, such tasks are often graded. The students are usually encouraged to discuss the assignments with classmates. However, many students lack the time or skills to solve these assignments or want to improve their grades [15, 18]. As a result, they copy code from other students, which is considered plagiarism [7, 15].

Detecting such cases of plagiarism manually becomes unfeasible at scale with an increasing number of submissions. Therefore, multiple plagiarism detectors like JPlag [29] and MOSS [32] were developed to automatically detect plagiarism in student assignments. They analyze the students' submissions and calculate a similarity score between each pair. Submission pairs with high similarity are associated with plagiarism and can be checked manually by a tutor or a teacher to decide if it is a case of plagiarism. Since students may modify or transform their copied code to try to bypass such detection mechanisms, plagiarism detectors must be able to deal with such measures to a certain degree [17].

The most widely used and state-of-the-art technique is token-based plagiarism detection [25]. Before comparison, the source code gets mapped to an alternative representation, the so-called token list, in a process called token abstraction. Afterward, an algorithm to calculate similarities between pairs of submissions gets applied to the token list. This token abstraction employs some abstraction from the representation of the code. The abstraction allows the comparison of the structure of the source codes, as this is more difficult to change when plagiarizing [17]. Therefore, it only extracts elements of the code's structure and abstracts from elements like variable names since they are not part of the code's structure and are easy to change. This abstraction improves plagiarism detection quality and resilience against modifications [18]. The mapping is often programming language specific, and we call the implementation of it a language module. The resulting tokens, as well as the calculation of similarities, are language-independent.

The most common way to support a new programming language is to create a new language module for that particular language. The reason for this is the assumption that having a dedicated language module for a specific language improves plagiarism detection since the token abstraction refers directly to the language. However, this process can be very time-consuming, yet it is unclear if this actually improves plagiarism detection. Therefore, it still needs to be determined if a dedicated language module for each programming language is even necessary. Our thesis will compare dedicated language modules with a language module supporting several programming languages. This allows us to draw conclusions that indicate whether a dedicated language module for each programming language is necessary or if a language module supporting multiple programming languages is sufficient. Furthermore, we want to derive takeaways for designing new language modules based on our approach.

To accomplish that, we create a language module for an intermediate representation (IR) supporting several programming languages. An IR is the code used internally by compilers to represent source code. We create the language module for the intermediate representation of LLVM [19]. LLVM is a collection of modular compiler and toolchain technologies that operate on their intermediate representation, LLVM IR. There are many frontends for programming languages that allow compiling source code to the LLVM IR, like the Clang compiler [30] for C and C++. Therefore, our language module supports multiple heterogeneous programming languages. We compare our LLVM IR language module with dedicated language modules for Java and C/C++ to assess their necessity, indicating whether dedicated language modules are generally necessary. In addition, we describe the design decisions behind it to derive conclusions for creating new IR language modules.

We evaluate our approach by implementing the LLVM IR language module for JPlag [29] and comparing it with the existing dedicated Java and C/C++ language modules from JPlag. Our evaluation is divided into two parts. We first want to evaluate the LLVM IR language module itself by comparing different designs and configurations with each other. This step will result in a suitable language module and the design decisions behind it. In the second part, we compare the LLVM IR language module with the dedicated Java and C/C++ language modules from JPlag regarding their plagiarism detection quality. From this, we can infer conclusions that indicate whether having a dedicated language module for each programming language is necessary and, with the first part, how new IR language modules can be designed.

Our results show that the two dedicated language modules perform better at detecting plagiarism in student submissions than our IR language module. This suggests that it is preferred to have language modules for each programming language as it improves plagiarism detection in student submissions. However, our IR language module shows better resilience to obfuscation attacks than the dedicated Java language module. As students can use frameworks to obfuscate their plagiarized code automatically [8], an IR language module can help detect such plagiarism. Regarding designing IR modules, we show that higher optimization levels can lead to better detection quality and that more fine-grained tokens can also improve plagiarism detection.

The remainder of this thesis is structured as follows. In chapter 2, we present the foundations for this work. Chapter 3 gives an overview of relevant work in this field. In chapter 4, we describe the concept of this thesis, followed by the construction of our LLVM IR language module in chapter 5. Chapter 6 contains the results of our evaluation. Chapter 7 presents ideas and suggestions for future work, and chapter 8 concludes this thesis.

2. Foundations

This chapter describes the fundamental concepts and software relevant to this thesis. Section 2.1 introduces token-based Plagiarism detection, which is relevant to the software plagiarism detection we focus on in this work. Then, section 2.2 focuses on the basics of the software we extend in this work, namely JPlag. Section 2.3 provides an introduction to attacks on plagiarism detection. Finally, section 2.4 overviews relevant concepts of LLVM and the LLVM IR.

2.1. Token-based Software Plagiarism Detection

Plagiarism can be defined as "the act of imitating or copying or using somebody else's creation or idea without permission and presenting it as one's own" [18]. Plagiarism in student assignments and programs is frequent in academic contexts, especially in computer science [6]. It is not intended that students submit solutions they did not create themselves or include significant work from others. A plagiarized program can hereby be defined as "a program which has been produced from another program with a small number of routine transformations" [27]. As a countermeasure to software plagiarism, many different plagiarism detection tools and algorithms are available today [25].

Token-based software plagiarism detection is the most common technique in academia [25]. The principle is that the language module extracts tokens from the code that represent the code's structure, which are then compared. Therefore, submissions with the same structure result in the same token list, regardless of their representation. In the first step, the source code of the submission files gets parsed and mapped to the token list. The mapping abstracts from the code's representation and is typically specific to a language. We call this token abstraction. Therefore, different programming languages require different token abstractions. Designing the token abstraction requires deciding the token types and defining rules for extracting the tokens from the parsed code. Consequently, the language module has two tasks: Parsing the source code of the submissions and extracting tokens according to specific rules. The idea is that the abstraction through token lists improves performance and resilience against modifications while preserving the essence of the program. Therefore, token abstraction is an essential part of the quality of plagiarism detectors. The second step is calculating a similarity score of two token lists and is mostly language-independent. The algorithms used in standard plagiarism detectors are greedy string tiling [34] in JPlag and fingerprinting [32] in MOSS. An example of a source code mapping to a token list is in Figure 2.1.

Source Code	**JPag Token**
`void printSum() {`	METHOD_BEGIN
`    int i = 1;`	VARDEF
`    int sum = 0;`	VARDEF
`    while (i <= 10) {`	WHILE_BEGIN
`        sum += i;`	ASSIGN
`        System.out.println(sum);`	APPLY
`        i++;`	ASSIGN
`    }`	WHILE_END
`}`	METHOD_END

Figure 2.1.: Example of JPlag's token abstraction to map Java source code to a token list

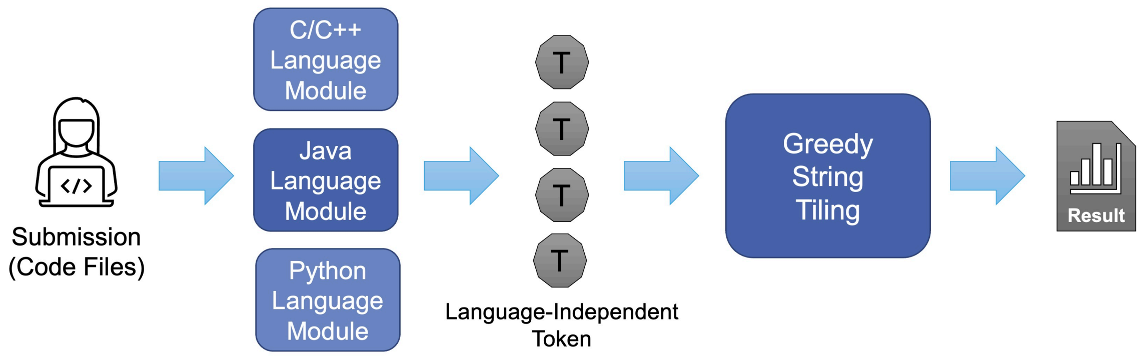

Figure 2.2.: JPlag Pipeline

2.2. JPlag

JPlag is an open-source token-based plagiarism detection tool developed in 1996 at the University of Karlsruhe [29]. It supports a variety of different programming languages [16]. To support a programming language, JPlag requires a language module that extracts a token list from the source files. The comparison of code submissions consists of two phases, as seen in Figure 2.2:

Token Abstraction

First, the corresponding language module takes the submissions and parses or scans each, creating an abstract syntax tree (AST). Every language module can use whatever method is preferred to build the AST. For example, the Java module uses the JavaC API, and the C/C++ module uses an ANTLR grammar. After this step, the AST was built that describes the program's structure. Afterward, a list of language-independent tokens gets extracted from the AST, where each token type stands for a syntactic concept of the language, like assignments or loops. Details describing semantics, such as variable types or method

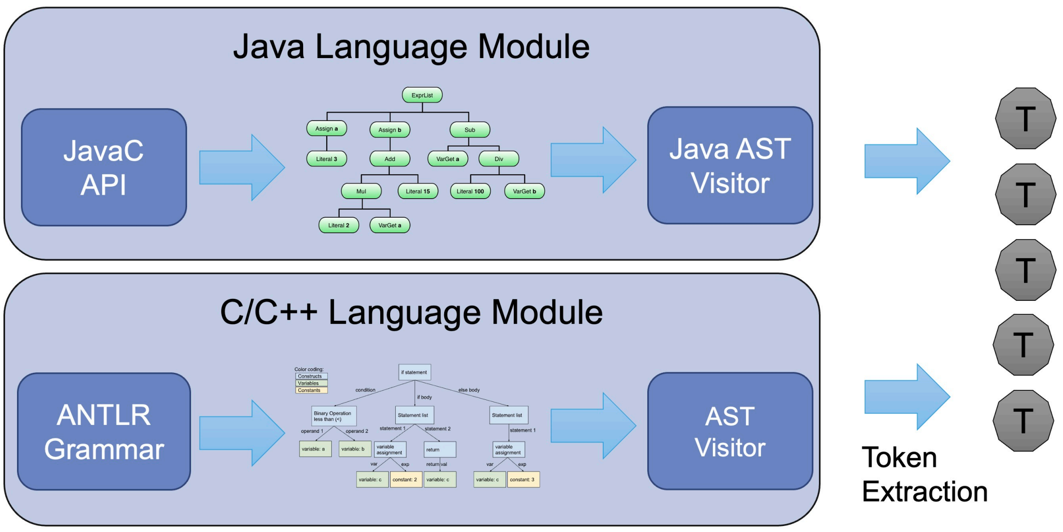

Figure 2.3.: JPlag's language module

signatures, are mainly abstracted so that the tokens represent the program's structure. That prevents many common code modifications, like parameter renaming. This phase, also seen in Figure 2.3, is highly language-dependent, and each language module can decide what types of tokens exist and how they get extracted.

Calculation of the Similarity Score

JPlag iterates over each pair of token lists and tries to cover one token sequence with sections of the other. The used algorithm is a modified version of greedy string tiling [34] and returns a list of matching sections within the two token lists. JPlag then outputs the similarity score, meaning how much could be matched as a percentage. This score represents the similarity of the two submissions. The minimum token match describes the minimum number of tokens that must be identical to get marked as a match and increase the similarity score. The optimal value can vary and depends on the language module and the submissions. The minimum token match prevents detecting brief identical sections typical in programs and do not indicate plagiarism. Additionally, JPlag allows the specification of a base code, which is ignored in matching every file and is meant for templates. The minimum token match and the base code are parameters that the user can set.

2.3. Obfuscation Attacks

As mentioned before, students commonly attempt to hide their plagiarism by modifying or transforming their submissions. If they explicitly intend to deceive a plagiarism detector, we refer to such an action as an obfuscation attack. Karnalim [17] defines eight levels of attacks on source code plagiarism detection and lists specific attack types. Therefore, when designing a language module, one has to consider which attack vectors are prevented by the design. This step is even more relevant with program transformation frameworks like MOSSAD [8]. They aim to automatically create one or multiple plagiarisms that can not be detected from one original submission. This can be achieved with obfuscation attacks, for example, by inserting new lines of code or reordering existing ones without affecting the program's results. In the case of JPlag, this splits up matching sections and keeps the length of the sections below the minimum token match and is therefore not accounted as plagiarism. Consequently, it is even more critical that plagiarism detectors are resilient to such obfuscation attacks to a certain degree.

2.4. LLVM

LLVM [23] is a collection of modular and reusable compiler and toolchain technologies. It began as a research project at the University of Illinois, providing a language and target-independent compiler framework. Designed for static and dynamic compilation by providing high-level information for compiler transformations at all stages of compilation [19]. LLVM can be used to build optimizing compilers for high-performance programming languages and compiler-based tools such as Just-In-Time (JIT) translators, analysis tools, and more. Initially, they achieved this through the LLVM virtual instruction set and a compiler design exploiting this code representation.

LLVM IR

LLVM IR is the intermediate representation of LLVM, a low-level virtual instruction set used as the internal code representation, as seen in Figure 2.4. It is a three-address code representation designed to be both language-independent and target-independent. The instruction set avoids machine-specific constraints such as physical registers and low-level calling conventions but captures the key operations of ordinary processors. LLVM provides an infinite set of typed virtual registers in Static Single Assignment (SSA) [1] form that requires each variable to be assigned exactly once and defined before it is used. These registers can hold values of primitive types like integers, floating points, booleans, and pointers. It provides a language-independent type system that enables a broad class of high-level transformations on low-level code, instructions for performing type conversions, and low-level address arithmetic while preserving type information. Further, it supports exception handling for implementing language-specific exception semantics and type-safe pointer arithmetic [19, 21].

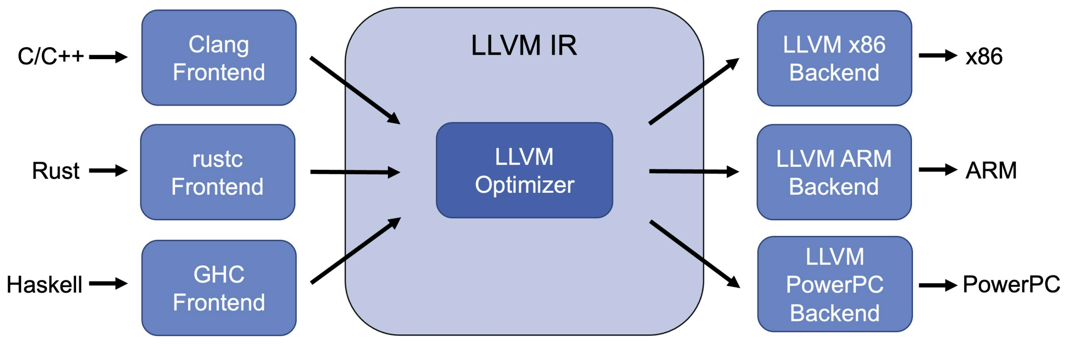

Figure 2.4.: LLVM Phases

Compiler Framework

The LLVM compiler framework allows sophisticated transformations at all stages of compilation by operating on the LLVM IR of a program. Its components include interprocedural optimizations at link time, machine-dependent optimizations at install time on each system, and dynamic optimization at run time. Furthermore, they contain static and dynamic analyses, native code generators, JIT compiler support, and more [19, 20].

Today, LLVM has become an umbrella project for several subprojects focusing on compiler and toolchain technologies.

3. Related Work

Much research has been done in source code plagiarism detection, resulting in many tools and algorithms [25]. There is further work focusing on token abstraction as well as the support of multiple programming languages.

Greule conducts two experiments in his bachelor's thesis [10] addressing the issue of token abstraction. In his first experiment, he analyzes the extraction of tokens on two different abstraction levels. He compares the use of a parser-based approach with a lexer-based approach. The parser-based method uses an AST, which extracts syntactical elements representing the structure, while the lexer-based approach uses a lexer to extract keywords, separators, and operators. In his second experiment, he analyzes token abstraction by defining categories for sets of token types and evaluates the quality of these categories. His thesis focuses on the design of token abstraction for language modules in general. Our thesis investigates whether these designs must be performed for each dedicated language module. We do this by focusing on an IR language module's design and token abstraction, resulting in support for multiple programming languages.

Karnalim [17] analyzes plagiarism detection on bytecode level by extracting tokens from Java bytecode. Using Java bytecode supports languages like Java, Kotlin, and Scala, which compile to it. During token extraction, the instructions get generalized and reinterpreted, linearizing the methods. Generalization combines several instructions into one token. For example, iadd (integer addition), ladd (long addition), fadd (float addition), and dadd (double addition) get mapped to only one addition token. Reinterpretation transforms special instructions into simpler ones. For example, tableswitch and lookupswitch instructions get translated into goto instructions. While the author does not provide details on the design decisions, we want to describe the design of the language module to derive takeaways for new language modules of IRs. Also, Karnalim works directly with the bytecode while we extract our tokens from an AST. Moreover, it supports only JVM-based languages, while our approach with LLVM IR supports more heterogeneous languages than Karnalims.

Rabbani and Karnalim [31] proposed a similar approach. They detect source code plagiarism on .NET programming languages by extracting tokens from a low-level representation. This allows plagiarism detection for languages like Visual Basic and C#. They compare the code in a pairwise approach. In the first step, both source codes get compiled to their .NET CIL executable file. Afterward, the executable files get disassembled using the .NET CIL disassembler to generate readable text of .NET CIL information. While the token extraction is done directly on the text, they describe it only briefly, demonstrating how it works. However, our thesis focuses on the token abstraction on an AST, describing the design and deriving takeaways for new IR language modules. Moreover, our approach supports more heterogeneous languages as well.

The plagiarism detection tool Dolos [24] is based on the tree-sitter parser library [4], which supports various programming languages. In the token abstraction step, Dolos uses this library to convert each source file into an AST. Afterward, each AST gets serialized into a list of tokens. Although Dolos supports various programming languages, the token extraction step is not explained further. In contrast, we want to explain how we extract our tokens to derive takeaways for new IR language modules.

Christian Arwin and S. M. M. Tahaghogi [3] propose the plagiarism detection tool XPlag, which can detect plagiarized source code between programming languages. First, all source files get compiled to an IR, namely the GCC Register Transfer Language (RTL), using the GCC compiler with compiler optimizations. Then, the RTL gets converted into tokens used to calculate the similarity. In the token extraction step, they represent the tokens as keywords of the RTL, which are filtered to avoid variable or register names. XPlag shows good results, as well as shows that the highest optimization level yields the best results. However, the token extraction is only described briefly by mentioning the relevant keywords. In contrast, we focus on token extraction and give a more detailed description. Moreover, XPlag works with keywords and supports only programming languages that can be compiled with GCC, while our approach works with an AST and supports a broader range of programming languages with LLVM IR.

Jhi et al. [14] propose a software plagiarism detection prototype based on the observation that some critical runtime values of a program are difficult to replace or eliminate by semantic preserving transformation techniques. In several steps, they generate a sequence of so-called core values, which then get compared. They extract runtime values, defined as values from the output operands of the executed machine instructions. Then, the core values get extracted from these runtime values. This sequence is then refined and can be compared to others to calculate the similarity of programs. In Jhi et al. [13], they improved this approach to become resilient to reordering attacks. While they explain the sequence extraction, the prototype only applies to programs that can be compiled with GCC. In contrast, our approach with LLVM IR supports a broader range. Furthermore, our token extraction works on an AST, so we compare the structures of the programs while their approach only compares runtime values.

Caldeira et al. [5] propose a method for clone detection using the LLVM IR. Clone detection is the detection of duplicated code fragments within the software that reduces maintainability. In their approach, the source code gets compiled to the LLVM IR beforehand. Afterward, a clone detection tool is used on the IR, detecting such code duplicates. They show that the compilation to LLVM IR reduces the syntactical variability of semantically identical code and improves code clone detection. The main difference from our thesis is that we are working on plagiarism detection instead of clone detection. Code duplicates arise unintentionally by reusing the same code fragments several times with minor modifications. In plagiarism detection, we deal with an attacker who intentionally uses obfuscation attacks to bypass plagiarism detectors.

Additionally, there is further work on software plagiarism detection using machine learning approaches. Yasaswi et al. [35] propose an unsupervised learning approach for detecting source code plagiarism in an IR. In their method, they extract static features from the IR of a program. They use the GCC compiler for the IR to compile the source code to the GCC IR. Afterward, they apply an unsupervised learning technique on the extracted features to form clusters of similar student solutions. Ullah et al. [33] proposed a different approach. In a preprocessing step, they convert the source code into tokens now containing frequency details. Afterward, they weight the tokens and perform a principle component analysis to extract features. Ultimately, a multinomial logistic regression model is applied to these features to classify the source codes based on predictions. Since their method does not look at the syntax of a program, their approach is programming language-independent. Both approaches support multiple programming languages simultaneously using machine learning techniques. However, we use the AST to represent the syntactic elements of the source code and generate tokens for the comparison step from it. We support multiple programming languages by allowing plagiarism detection on the intermediate representation of source codes.

4. Concept

Token-based plagiarism detection is a state-of-the-art technique frequently used in educational settings [25]. While the pairwise comparison of the token lists is language-independent, token abstraction is usually programming language-specific. As mentioned in chapter 1 and section 2.2, adding support for a new programming language usually involves designing and implementing a new language module specific to that language. However, this process can be very time-consuming. It must first be defined how the submission files get parsed, which is an effort of a technical nature. Then, the token abstraction must be defined, as it is different for each programming language, which is work at the conceptual level.

Moreover, there is no systematic process or pre-defined rules that explain how to design a new language module. Most design decisions are based on experience from existing language modules that have worked well so far. "As a rule, tokens should be chosen such that they characterize the essence of a program's structure (which is difficult to change by a plagiarist) rather than surface aspects." [29]. This makes it more challenging to implement language modules for new programming languages, which actually provide a desirable detection quality.

However, it still needs to be determined if having a dedicated language module for each new programming language is even necessary. A language module supporting multiple programming languages might provide similar results to the dedicated ones. This way, the parsing and token abstraction do not have to be redefined for each language. We aim to draw conclusions indicating whether having dedicated language modules for each programming language is necessary or if a language module supporting multiple programming languages is sufficient. Additionally, we want to derive takeaways for designing new language modules based on our approach.

To achieve this, we create a language module in an IR supporting several programming languages. To accomplish support for several programming languages, we compile the source code for each programming language to the IR. Afterward, we insert the IR code into our IR language module, generating the tokens for the comparison. Figure 4.1 shows an illustration of this.

As discussed in chapter 3, there is work on language modules in IRs. However, there is a lack of evidence regarding the design of the token extraction, as there is no systematic process that explains how to design a new language module. We compare our IR language module with dedicated language modules to assess their necessity. In addition, we describe the design decisions behind it to derive takeaways for IR language modules and to support

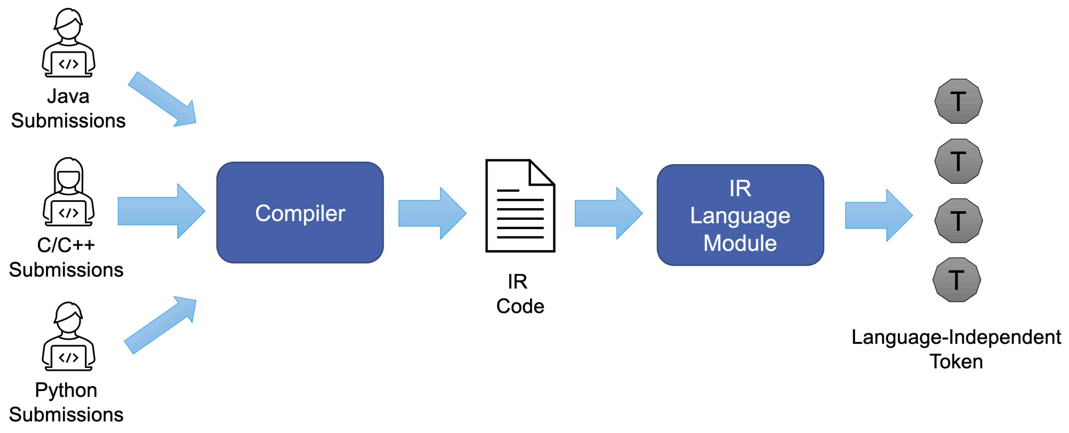

Figure 4.1.: Process to support multiple programming languages

the creation of new modules. These design decisions consist of the decisions behind the token abstraction and the configurations for the minimum token match and compiler optimizations.

Moving from dedicated to IR language modules adds a new abstraction level since the source code must be compiled to the IR. This process usually comes with a loss of information because only some elements get extracted, but some get abstracted. Additionally, this abstraction can vary with different compiler optimizations performed on the IR. We want to investigate if this loss of information affects detecting software plagiarism by comparing our IR language module with dedicated ones. If the IR languages module performs worse than the dedicated ones, we assume relevant information gets lost in the abstraction step.

We evaluate our approach by implementing a language module in JPlag [29] and comparing it with the existing dedicated language modules. We implement this language module for the LLVM IR. There, we focus on the token selection and the token extraction. The token selection consists of deciding which syntactical elements should be assigned a token. The token extraction describes when which tokens will get extracted.

Since the LLVM IR is extensive and complex, we propose two versions for the language module with different token abstractions. The first, simple version implements a simple token abstraction. It focuses only on a subset of the LLVM IR language, namely function and program instructions because they are the most relevant parts of the LLVM IR. The second, advanced version implements a more extensive token abstraction. It is more complex and considers the whole IR to take advantage of that. This differentiation lets us derive design ideas when comparing the different versions.

5. LLVM IR Language Module

In this chapter, we describe the creation of the LLVM IR language module. We list the token types and describe the mapping of LLVM IR elements to our token types. Furthermore, we explain the design decisions behind our language module. Section 5.1 describes the design of our simple version of the LLVM IR language module, and section 5.2 describes our advanced version.

5.1. Simple Token Abstraction

For the simple version of our language module, we describe a simple token abstraction. This means we decide on token types and generate tokens only for functions and program instructions because they are the most relevant parts of the LLVM IR.

5.1.1. Functions

Table 5.1 shows the token types we decided on for functions. We create a separate token for the beginning and the end of a function. This distinction allows us to separate individual methods from each other. Furthermore, we distinguish between a function definition and a function declaration since defined functions are implemented in the file, and declared functions are externally defined functions that just get called. We decided not to generate tokens for function parameters as they can be an easy target for an obfuscation attack. An attacker can circumvent plagiarism detection by adding an unused parameter to a method definition because one more token gets extracted. Additionally, we decided not to generate tokens for attributes of functions and parameters since they merely contain information relevant to plagiarism detection. They are primarily relevant for the compiler, for example, to declare how a parameter gets aligned in memory.

Category	**Token Type**	**Description**
Functions	FUNCTION_BODY_BEGIN	Start of a function
Functions	FUNCTION_BODY_END	End of a function
Functions	FUNCTION_DECLARATION	Declaration of a function
Functions	FUNCTION_DEFINITION	Definition of a function

Table 5.1.: Simple Function Token Types

Category	Token Type	Description
Terminator instructions	RETURN	Return control flow
Terminator instructions	BRANCH	Transfer control flow
Terminator instructions	CONDITIONAL_BRANCH	Conditional branch statement
Terminator instructions	INVOKE	Transfer control flow to function
Terminator instructions	CALL_BRANCH	Call branch statement
Terminator instructions	SWITCH	Switch branching statement
Terminator instructions	INDIRECT_BRANCH	Indirect branch statement
Terminator instructions	RESUME	Resume from a function call
Terminator instructions	CATCH_SWITCH	Set of possible catch handlers
Terminator instructions	CATCH_RETURN	Ends an existing exception
Terminator instructions	CLEAN_UP_RETURN	Transfer control flow

Table 5.2.: Simple Terminator Token Types

5.1.2. Terminator Instructions

Table 5.2 shows the token types for terminator instructions. We have a token type for each terminator instruction of the IR for the simple version except one. The reason for separate token types is that different control structures use different instructions. For example, an *if* expression in a higher-level language is mainly implemented with a conditional branch, while a *switch case* is implemented with a switch instruction. Furthermore, instructions, like invoke and resume, are essential for the exception-handling routine of the LLVM IR, which results from exception handling in the source code. The terminator instruction, which is not represented, is the unreachable instruction, which specifies that a part of the code is unreachable. We do not generate a token for this to improve the resilience of obfuscation attacks that add unreachable code.

5.1.3. Operations

General operations are represented with only one token type, as seen in Table 5.3. This operation token type combines multiple instructions, like instructions for adding integers and floats, subtracting integers and floats, shifts, and binary instructions. The idea is that these instructions often get generated from assignments like this:

```
int x = 4 + 5;
double y = 2.5 * x;
int z = x << 2;
```

The existing language modules in JPlag map these assignments on the same token independent of the operator used. To stay consistent with these, we map such instructions on the same token type for the simple version. We do this by generating an OPERATION token for each of the instructions.

Category	Token Type	Description
Operations	OPERATION	Operation instructions

Table 5.3.: Simple Operation Token Types

Category	Token Type	Description
Memory Operations	ALLOCATION	Allocates new memory
Memory Operations	LOAD	Loads a value from memory
Memory Operations	STORE	Stores a value in memory
Memory Operations	FENCE	Introduce Happens-before relation
Memory Operations	COMPARE_EXCHANGE	Atomic exchange of values
Memory Operations	ATOMIC_RMW	Atomic read, modify and write
Memory Operations	GET_ELEMENT_POINTER	Get pointer from an element

Table 5.4.: Simple Memory Token Types

5.1.4. Memory Operations

Table 5.4 shows the token types for memory operations. For the simple version, we have a token type for each memory operation of the IR. Because of this, we can differentiate between the different memory operations as they are fundamentally different.

5.1.5. Other Operations

The LLVM IR also specifies other instructions for various use cases. These are instructions for comparisons, conversions, exception handling, and more. Table 5.5 shows the token types we decided on for the simple version.

Regarding comparisons, the LLVM IR provides instructions for integer and float comparisons. Different token types would enable an easy obfuscation attack by changing integers to floats or vice versa, thus generating different tokens. So, we map both instructions on one COMPARISON token.

For conversions, the LLVM IR consists of several different instructions, like converting integers to pointers. Changing the type of variables like integer to float would be an attack vector for an obfuscation attack similar to the comparisons. As a result, we map all conversions on one CONVERSION token.

5.2. Advanced Token Abstraction

The advanced version of the LLVM IR language module considers the whole IR. This allows us to derive whether other aspects of the IR than functions and instructions are

Category	Token Type	Description
Conversions	CONVERSION	Conversions and casts
Comparisons	COMPARISON	Comparison of integers and floats
Others	PHI	SSA phi instruction
Others	SELECT	Selects value based on condition
Others	FREEZE	Stop propagation of particular values
Others	CALL	Function call
Others	VARIABLE_ARGUMENT	Instruction for variable size of arguments
Others	LANDING_PAD	Landing pad for an exception
Others	CATCH_PAD	Catch handler for an exception
Others	CLEAN_UP_PAD	Cleanup from an exception

Table 5.5.: Simple Other Token Types

Category	Token Type	Description
Files	FILENAME	Name of the LLVM IR file
Functions	FUNCTION_BODY_BEGIN	Start of a function
Functions	FUNCTION_BODY_END	End of a function
Functions	BASIC_BODY_BEGIN	Start of a basic block
Functions	BASIC_BODY_END	End of a basic block
Functions	FUNCTION_DECLARATION	Declaration of a function
Functions	FUNCTION_DEFINITION	Definition of a function

Table 5.6.: Advanced Token Types for Functions and Files

relevant and how we can use them. We grayed out all token types from the simple version to better distinguish between new token types and types present in the simple version.

5.2.1. Functions

Table 5.6 shows the token types we created for functions and files. The difference from the simple version is that we have a token type for the file name. This token type represents the start of a file, as the filename is always at the top of the IR code. Additionally, we have token types for the beginning and end of a basic block, which is a sequence of instructions that terminates with an instruction that changes the program's control flow. This distinction allows differentiating programs with different control flows because different tokens will be extracted.

5.2.2. Globals

We have new token types representing global elements of the IR, as seen in Table 5.7. Global variables and type definitions are relevant parts of the functionality of an LLVM IR

Category	Token Type	Description
Globals	GLOBAL_VARIABLE	Definition / declaration of global variables
Globals	ASSEMBLY	Inline assembly and assembly module
Globals	TYPE_DEFINITIONS	Definition of a type
Constants	STRUCTURE	Definition of a struct
Constants	ARRAY	Definition of an array
Constants	VECTOR	Definition of a vector

Table 5.7.: Advanced Token Types for Global Elements

Category	Token Type	Description
Terminator instructions	RETURN	Return control flow
Terminator instructions	BRANCH	Transfer control flow
Terminator instructions	SWITCH	Switch branching statement
Terminator instructions	CASE	Case statement for a switch
Terminator instructions	CONDITIONAL_BRANCH	Conditional branch
Terminator instructions	INVOKE	Transfer control flow to function
Terminator instructions	CALL_BRANCH	Call branch statement
Terminator instructions	RESUME	Resume from a function call
Terminator instructions	CATCH_SWITCH	Set of possible catch handlers
Terminator instructions	CATCH_RETURN	Ends an existing exception
Terminator instructions	CLEAN_UP_RETURN	Transfer control flow

Table 5.8.: Advanced Terminator Token Types

program and, therefore, are assigned a token type. We map definitions and declarations for global variables on a single token type since differentiating between them is a possible attack vector. For example, declaring a global variable and assigning it a value later is simple but generates different tokens. Furthermore, we have token types for constant aggregate types. Classes and structures in high-level languages get mapped to structures in the IR. Since initializing them and other aggregates like vectors and arrays are integral to a program, we also define token types for them.

5.2.3. Terminator Instructions

Table 5.8 shows our token types for terminator instructions. The difference to the simple version is that the INDIRECT_BRANCH token type was removed because it is similar to the standard branch. Differentiating between them is a possible attack vector for obfuscation attacks. So, we extract the BRANCH token for the indirect branch instruction, too. Second, we have a new token type CASE for the cases of a switch statement. This allows tokens for the case statements of higher-level languages to represent different outcomes.

Category	Token Type	Description
Binary Operations	ADDITION	Addition
Binary Operations	SUBTRACTION	Subtraction
Binary Operations	MULTIPLICATION	Multiplication
Binary Operations	DIVISION	Division
Binary Operations	REMAINDER	Remainder
Bitwise instructions	SHIFT	Shift
Bitwise instructions	AND	And
Bitwise instructions	OR	Or
Bitwise instructions	XOR	Xor
Vector operations	EXTRACT_ELEMENT	Extract an element
Vector operations	INSERT_ELEMENT	Insert an element
Vector operations	SHUFFLE_VECTOR	Shuffle a Vector
Aggregate Operations	EXTRACT_VALUE	Extract a value
Aggregate Operations	INSERT_VALUE	Insert a value

Table 5.9.: Advanced Operation Token Types

5.2.4. Operations

The difference from the simple version for general operations is that we now have token types for the different operations, as seen in Table 5.9. This distinction allows us to distinguish between them and have a more fine-grained token extraction.

Regarding the operations, the LLVM IR differentiates between operations on floats and integers. We map these on the same token type to prevent obfuscation attacks by simply changing the type. For example, the addition of integers and the addition of floats gets mapped on the ADDITION token type. This measure increases resilience towards obfuscation attacks as changing the variable type from integer to float should not decrease similarity.

5.2.5. Memory Operations

Table 5.10 shows the token types for memory operations. The difference to the simple version is that we have a new token type, ATOMIC_ORDERING, allowing us to represent synchronizations in more detail.

5.2.6. Other Operations

The LLVM IR also specifies other instructions for various use cases. Table 5.11 shows the token types for these. The difference to the simple version is that we removed the token type for the freeze instruction, as the instruction is relevant for the compiler but not for detecting plagiarism. Furthermore, we have another token type, BITCAST, to distinguish

Category	Token Type	Description
Memory Operations	ALLOCATION	Allocates new memory
Memory Operations	LOAD	Loads a value from memory
Memory Operations	STORE	Stores a value in memory
Memory Operations	FENCE	Introduce Happens-before relation
Memory Operations	COMPARE_EXCHANGE	Atomic exchange of values
Memory Operations	ATOMIC_RMW	Atomic read, modify and write
Memory Operations	ATOMIC_ORDERING	Ordering for synchronization
Memory Operations	GET_ELEMENT_POINTER	Get pointer from an element

Table 5.10.: Advanced Memory Token Types

Category	Token Type	Description
Conversions	BITCAST	Casts between types
Conversions	CONVERSION	Conversions and casts
Comparisons	COMPARISON	Comparison of integers and floats
Others	PHI	SSA phi instruction
Others	SELECT	Selects value based on condition
Others	CALL	Function call
Others	VARIABLE_ARGUMENT	Instruction for variable size of arguments
Others	LANDING_PAD	Landing pad for an exception
Others	CLAUSE	Clause statement for landing pad
Others	CATCH_PAD	Catch handler for an exception
Others	CLEAN_UP_PAD	Cleanup from an exception

Table 5.11.: Advanced Other Token Types

conversions further. While the other conversion operations describe operations between built-in types like integer to pointer, the bitcast instruction describes conversions between all other types. This instruction is essential for allocating memory and representing structures from higher-level programming languages. Therefore, we give it a separate token type to distinguish it from the other conversions. Lastly, we added a token type CLAUSE, which is relevant for exception handling as it is part of the catch statement in higher-level languages.

As with the simple version, the LLVM IR provides instructions for integer and float comparisons. So, we map both instructions on one COMPARISON token again. Regarding conversions, just like the simple version, changing the type of variables, like integer to float, would be an attack vector for an obfuscation attack. As a result, we map all conversions except the bitcast instruction on one CONVERSION token again.

5.2.7. Meta Data, Types and Other Elements

As already mentioned, the LLVM IR is quite extensive. It allows metadata to be attached to instructions and global objects that can convey extra information about the code to the optimizers and code generators. However, the metadata does not contain any relevant information for plagiarism detection. Therefore, we do not have any token types representing it and do not generate any tokens for it. We additionally refrained from generating tokens for types in the LLVM IR as they can be an easy target of obfuscation attacks by changing the types of variables in the source code. There are also other elements of the LLVM IR, which we do not cover here, like the data layout, compiler-specific code, visibility styles, calling conventions, and garbage collection strategies. We do not generate any tokens for them either, as we do not see their relevance for detecting plagiarism.

6. Evaluation

This chapter contains the evaluation results. We discuss the LLVM IR language module and compare our language module to the dedicated Java and C/C++ language modules. We describe the methodology of our evaluation in section 6.1. In section 6.2, we present our results and conduct a discussion. Section 6.3 lists the limitations of our evaluation, and section 6.4 describes threats to validity.

6.1. Methodology

In this section, we describe the methodology of our evaluation. We explain the implementation details of our LLVM IR language module to extend JPlag. We state our approach to assess the necessity of dedicated language modules and how we derive conclusions for new IR language modules. Afterward, we show the GQM-Plan of our thesis and describe the data sets and their modifications we use for our evaluation.

6.1.1. Implementation

We implement our LLVM IR language module using JPlag to conduct our evaluation.

For the parsing of the input LLVM IR code files, we use ANTLR [28]. ANTLR is a parser generator for reading, processing, executing, or translating structured text or binary files. With a LLVM IR grammar, we can generate an AST, a parser, and an AST listener interface. We use The LLVM IR grammar from the ANTLR grammar repository [2], with the most recent modification in August 2023[1].

We extract the tokens from the AST with the ANTLR listener interface. This allows us to generate a token for entering each node of the AST representing a syntactic element of the IR. The tokens are generated by their matching listener method. As an example the *enterFunctionDefinition* listener method always generates a FUNCTION_DEFINITION token.

The listener interface distinguishes further between instructions on values and expressions, which are instructions on constants. For example, the listener interface contains a method *enterAddExpr* for adding expressions and a method *enterAddInst* for adding values. In our implementation, we do not distinguish between them and generate for instructions

[1]https://github.com/antlr/grammars-v4/tree/768b12e1db509aa700a316e3eed1e23e8c4bdb06/llvm-ir

the same tokens as for their expressions. This avoids obfuscation attacks by changing variables and constants in the source code to generate different tokens.

6.1.2. Approach

For the evaluation, we compare our LLVM IR language module with dedicated Java and C/C++ language modules to determine if the abstraction to an IR results in a loss of relevant information. With this knowledge, we derive conclusions that indicate whether we need dedicated language modules or if an IR module supporting multiple programming languages is sufficient or even better. Additionally, we explain the design decisions behind our language module to derive takeaways that support the creation of new IR language modules.

The data sets on which we carry out our tests can be seen in subsection 6.1.3. To use the data, we compile the C/C++ source code to LLVM IR with the Clang compiler[2] [30]. The Java source files get compiled to LLVM IR with GraalVM[3] [9]. For the evaluation, the LLVM IR code is then used as input for JPlag, using the LLVM IR language module to generate the similarities. The generation of the similarities for the dedicated language modules will be done by directly running JPlag with the existing language modules on the respective data. Here, we will use the Java language module for the Java source code files and the ANTLR-based CPP language module for the C/C++ files.

We conduct our evaluation by comparing chosen metrics for the different language modules and configurations with each other. For the comparison, we calculate several metrics for the difference of similarities between unrelated submission pairs and plagiarism pairs. We do this because a larger difference between the similarities of non-plagiarized and plagiarized pairs is more desirable. The further plagiarism is separated from non-plagiarism, the easier it is for the user to recognize it. The score, which is the sum of all metrics, determines the plagiarism detection quality. We use the following three values as metrics, with a higher value being better:

- Difference of the mean between plagiarized submission pairs and unrelated pairs as $Diff_{Mean}$
- Difference of the median between plagiarized submission pairs and unrelated pairs and as $Diff_{Median}$
- Difference between the 25% quantile of plagiarized pairs and the 75% quantile of unrelated submission pairs as $Diff_{Quantile}$

Therefore, a higher score means a better plagiarism detection quality.

The evaluation of the LLVM IR language module is divided into two parts, which are also the two goals of our GQM-Plan in Table 6.1. The first part consists of a quality analysis of the LLVM IR module, where we evaluate the module's different token abstraction versions

[2]Version 15.0.7-arm64-apple-darwin22.0
[3]Version GraalVM CE 22.3.2

Goal	Question	Metric
1. Analyze the different token abstraction versions and configurations of the LLVM IR language module	1.1 Which token abstraction version results in a better detection quality?	1.1.1 Similarity difference of the LLVM IR language module with the simple and advanced version
	1.2 What minimum token match provides the best result?	1.2.1 Similarity difference of the LLVM IR language module with different minimum token matches
	1.3 What optimization level provides the best result?	1.3.1 Similarity difference of the LLVM IR language module with different optimization levels
2. Analyze whether a language module in an IR performs as well as dedicated language modules in plagiarism detection	2.1 How does moving to an IR affect the plagiarism detection quality of a language module?	2.1.1 Similarity difference of the Java and C/C++ language module and the LLVM IR language module
		2.1.2 Change of the false positive rate comparing the Java and C/C++ language module with the LLVM IR language module

Table 6.1.: GQM-Plan

and configurations. There, we want to know which language module version, minimum token match, and optimization level provide the best results regarding our metrics. In the second part, we compare the language module with the dedicated C/C++ and Java language modules. We want to determine how moving to an IR affects plagiarism detection regarding the similarity difference between plagiarized and non-plagiarized pairs.

The first part shows us which is the better LLVM IR language module and configuration based on our metrics. In addition, we derive takeaways for new IR language modules by explaining the design decision behind the better language module. The result of the second part indicates whether dedicated language modules are actually necessary. If the dedicated language modules show better results than our IR language module, we conclude that compiling to an IR results in a loss of relevant information. This indicates that dedicated language modules are superior and necessary for achieving the best plagiarism detection quality. If the LLVM IR language module shows similar or better results, we conclude that code can be compiled to an IR with little or no loss of relevant information. This indicates that dedicated language modules are unnecessary, and IR language modules are superior because they allow support for multiple programming languages.

6.1.3. Data Sets

We use real data sets of student submissions to evaluate our LLVM language module and estimate quality in real-world scenarios. In this section, we describe the data sets we used and the modifications made to them.

6.1.3.1. Programming Homework Data Set for Plagiarism Detection C/C++

For C and C++, we use the Programming Homework Dataset for Plagiarism Detection [22]. This data set contains homework submitted by students during two introductory programming courses, where each course consists of multiple different assignments. The solutions were programmed using C and C++. It includes additional files about the groups of students considered to have been involved in plagiarism. This decision is based on code similarity.

We applied several modifications to the data set. In the first step, we removed all files named *.c* and *.cpp* because with no file name, they were not in the ground truth and, therefore, are unlabeled. We further removed the course B2017 because it contains no plagiarized submissions. Moreover, many files contain information about which course the file belongs at the beginning of the file. Since these descriptions were not valid C/C++ code, we had to remove them to compile them. Some source codes also contain special characters like U+0001, U+0002, and null characters, which we removed as well to compile the codes. Additionally, we removed all inline and block comments because this is important for a later step, and they would only get ignored later. Therefore, we checked every line of each file and deleted the line if it started with a forward slash or was part of a block comment. These modifications affected 12943 files. Many submissions besides comments only contain an empty main method or variables that were never used. These pairs were not marked as plagiarism but had high similarities. To exclude such files from our evaluation, we removed all files being smaller or equal to 200 Byte, which affected 6459 files. We previously removed the comments because they increased the file size without containing relevant information. Next, since not every file could be compiled to LLVM IR because of syntax errors, we only use the files that could be compiled. So, for an equal evaluation, we removed the corresponding C/C++ files for which we could not generate the LLVM IR code. This affected 1915 files. We further renamed one variable in 10 files to avoid parsing errors with the CPP language module. In the last step, we removed 11 tasks from the three courses as they no longer contain plagiarized submissions.

The course A2016 is summarized as follows:

- **Language:** C
- **Submissions:** 6674 originally, 6569 after modifications
- **Plagiarized Pairs:** 863
- **Code length (min/avg/max):** 13/73.56/1402

The course A2017 is summarized as follows:

- **Language:** C
- **Submissions:** 13532 originally, 8911 after modifications
- **Plagiarized Pairs:** 389
- **Code length (min/avg/max):** 11/75.05/705

The course B2016 is summarized as follows:

- **Language:** C++
- **Submissions:** 12196 originally, 6204 after modifications
- **Plagiarized Pairs:** 45
- **Code length (min/avg/max):** 11/160.49/1268

6.1.3.2. Programming Homework Data Set for Plagiarism Detection LLVM IR

To generate our LLVM IR files, we compiled each C/C++ source file of our modified data set to LLVM IR. For the compilation, we use the Clang compiler [30] for C and the Clang++ compiler for C++. Since not every file could be compiled to LLVM IR because of syntax errors, we only use the files that could be compiled.

The data set is summarized as follows:

- **Course A2016 Code length (min/avg/max):** 58/410.71/4917
- **Course A2017 Code length (min/avg/max):** 42/432.95/4307
- **Course B2016 Code length (min/avg/max):** 14/9406.68/30029

We use the LLVM optimizer[4] to generate our optimized LLVM IR code. We generate it for the four optimization levels: O0, O1, O2, and O3. Table 6.2 shows a description of the code.

For our evaluation, we aggregate the results of all tasks for each course. To achieve this, we compute the similarities for each task of every course and put all similarities of non-plagiarized pairs together for each course. We do the same with the similarities of the plagiarized pairs, which leaves us with all the similarities for each course separated into non-plagiarized and plagiarized pairs. Figure A.2 shows a similarity distribution for each task with the advanced version of the IR language module.

6.1.3.3. PROGpedia Java

We use the PROGpedia [26] data set for Java source codes, a collection of tasks containing submissions from introductory programming courses. It consists of multiple programming exercises and the students' solutions. We use the PROGpedia-19 and PROGpedia-56 task

[4]Version 15.0.7-arm64-apple-darwin22.0

	A2016			A2017			B2016		
Data	**Min**	**Avg**	**Max**	**Min**	**Avg**	**Max**	**Min**	**Avg**	**Max**
Level O0	58	410.71	4917	42	432.95	4307	14	9406.68	30029
Level O1	59	412.59	4919	42	434.10	4291	14	9355.43	29946
Level O2	59	412.59	4919	42	434.10	4291	14	9355.43	29946
Level O3	59	412.59	4919	42	434.10	4291	14	9355.44	29946

Table 6.2.: LLVM IR Code lengths with different optimization levels for the Programming Homework data set

solutions from this data set. The PROGpedia-19 task does not contain a ground truth about whether a solution is plagiarized. Therefore, one of the advisors with experience in plagiarism detection in programming assignments labeled one of the Java data sets, using JPlag and manually verifying plagiarism pairs. This process has been done by running JPlag with the existing Java language module and going through high-similarity pairs to identify actual plagiarism. Additionally, the usage of a template in this data set has been identified. The template was then extracted and used as a base for our evaluation to reduce the number of false positives. For the PROGpedia-56 task, we used a modified version where all plagiarized pairs got removed first, leaving only non-plagiarized submissions. Afterward, plagiarized code was automatically generated for those submissions by inserting new statements or reordering existing statements. This allows us to evaluate the resilience of the language module to obfuscation attacks. The data set containing the submissions for the two tasks can be described as follows:

PROGpedia-19

- **Language:** Java
- **Submissions:** 66
- **Plagiarized Pairs:** 91
- **Code length (min/avg/max):** 74/141.38/294

PROGpedia-56

- **Language:** Java
- **Submissions:** 112
- **Plagiarized Pairs:** 168
- **Code length (min/avg/max):** 40/109.88/503

6.1.3.4. PROGpedia LLVM IR

To generate our LLVM IR files from the PROGpedia data set, we first compile each Java source file to Java class code using the Java compiler. Afterward, we use GraalVM [9] to generate LLVM bitcode files from the Java class code. Next, we use the LLVM disassembler from LLVM to generate our IR code files from the LLVM bitcode files. The resulting IR

PROGpedia-19 Task				PROGpedia-56 Task			
Optimization	**Min**	**Avg**	**Max**	**Optimization**	**Min**	**Avg**	**Max**
Level O0	2181	4475.18	6726	Level O0	1181	2491.34	8263
Level O1	1883	3522.82	5127	Level O1	955	1958.38	6281
Level O2	1879	3547.30	5186	Level O2	954	1954.49	6195
Level O3	1897	3554.09	5257	Level O3	983	1978.83	6260

Table 6.3.: LLVM IR Code lengths with different optimization levels for the PROGpedia data set

code has an average of 2 million lines. This causes performance problems with JPlag because of the long files and issues with plagiarism detection. Most of the generated code consists of internal Java functions compiled to LLVM IR and is independent of the actual written source code. Since this code is in every IR file, all files have a similarity of around 97%-99% without base code. To prevent this, we generate separate bitcode files for every function of a file with GraalVM from the Java class code. Then, we disassemble every bitcode function to IR code and filter these functions by only taking matching functions from the Java source code. We could significantly increase the performance and improve the detection quality with our function filtering, as Figure A.1 shows. The following shows a description of the IR code:

- **PROGpedia-19 Code length (min/avg/max):** 2189/4521.12/6782
- **PROGpedia-56 Code length (min/avg/max):** 1189/2511.05/8279

We again use the LLVM optimizer[5] to generate our optimized LLVM IR code. We generate it for the four optimization levels: O0, O1, O2, and O3. Table 6.3 shows a description of the code.

6.2. Results and Discussion

In this section, we present the results of our evaluation and discuss them. We show the best minimum token matches for the data sets and analyze them. Next, we show the results of the token abstraction versions and derive conclusions for the design of new IR language modules. Afterward, we show the impact of the optimization levels and make recommendations on using them. In the end, we provide the results for the comparison with the dedicated language modules and discuss their necessity.

[5]Version 15.0.7-arm64-apple-darwin22.0

6.2.1. Impact of the Minimum Token Match

We provide the results of the different minimum token matches and show the best value for each data set. We further analyze the best values and show how a good reference can be calculated.

Programming Homework Data Set

The results of the LLVM IR language module with different minimum token matches using the Programming Homework data set can be seen in Figure 6.1. The graphic shows the values of the three metrics for different minimum token matches. The best minimum token match is the value with the highest score, which is the sum of all three metrics. We get that the best value for the advanced version for course A2016 is 71. For course A2017, 61 is the best value, and for course B2016, it is 1043.

PROGpedia Data Set

Figure 6.2 shows the results of the different minimum token matches with the PROGpedia data set. It shows the values of the three metrics for different minimum token matches. The best value for the minimum token match of the advanced version is 189 for the PROGpedia-19 task because it has the highest score. For the PROGpedia-56 task, the best value of the advanced version is 101.

Best minimum token match

In Table 6.4, we can see the best minimum token matches for the different configurations. We used the advanced version of the language module for the optimization levels. The minimum token matches of the LLVM IR module are significantly higher than the minimum token matches of the Java and the CPP language modules. This is because a single line of source code is usually mapped to multiple lines of IR code as the IR code is more low-level. Subsection 6.1.3 shows that the average lines of codes of the IR are much higher than for the source code. For example, the average Java code length of the PROGpedia-19 task is 141.83 lines, while the average code length for the respective IR code is 4521.12 lines. Similarly to the dedicated language modules, we generate at least one token for most lines, so we need a much higher minimum token match to compensate for this. This also accounts for the difference between the simple and the advanced versions. The advanced version has a larger token set with finer-grained types of tokens. Therefore, it generates more tokens than the simple version, as seen in Table A.1, and needs a higher minimum token match.

Finding a good value for different data sets can be difficult because the range of possible candidates is much larger than for dedicated language modules. Values can range between 60 and 70 for code compiled from C to more than 1000 for code compiled from C++.

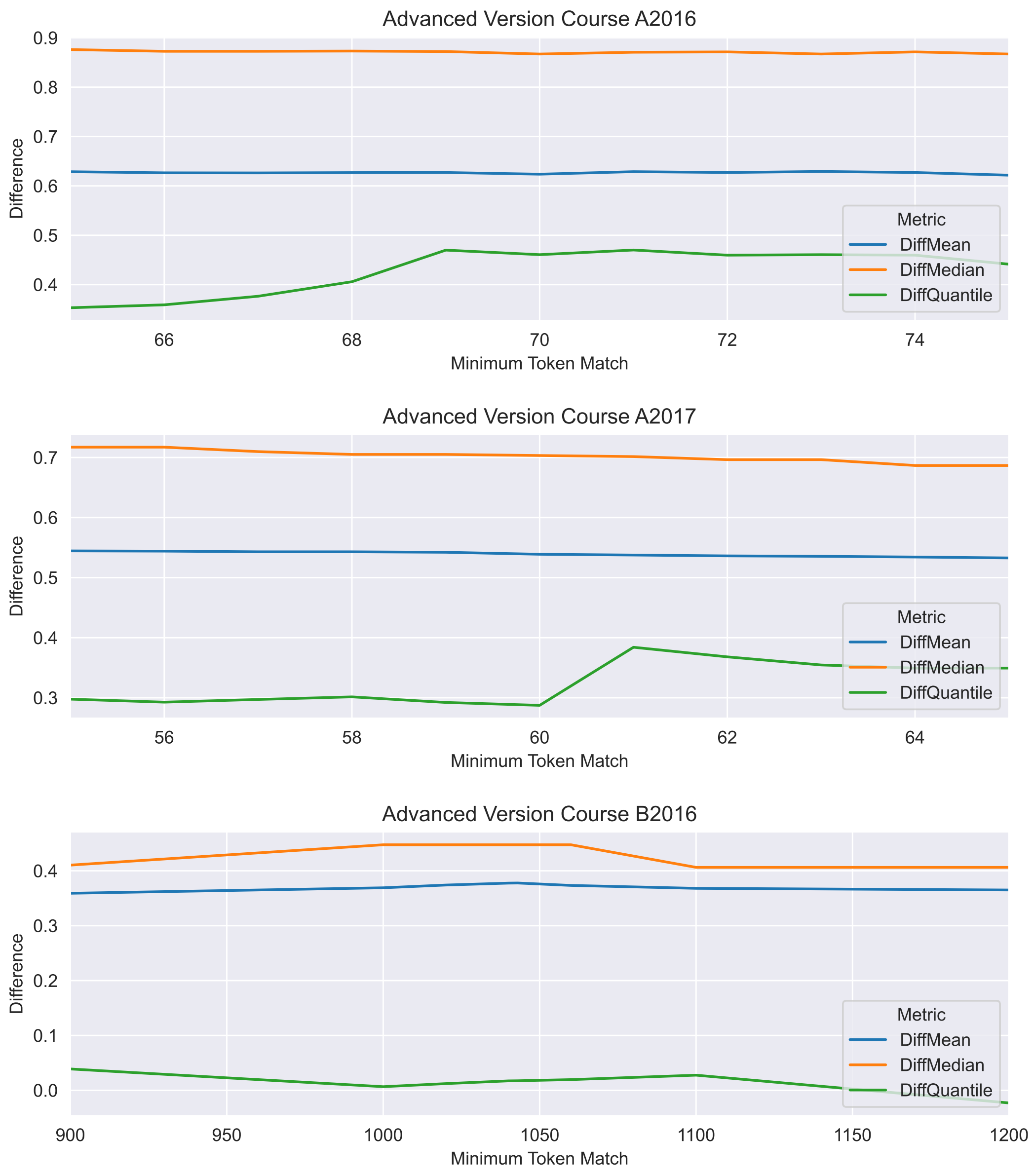

Figure 6.1.: Metrics for different minimum token matches using the advanced language module for the Programming Homework data set

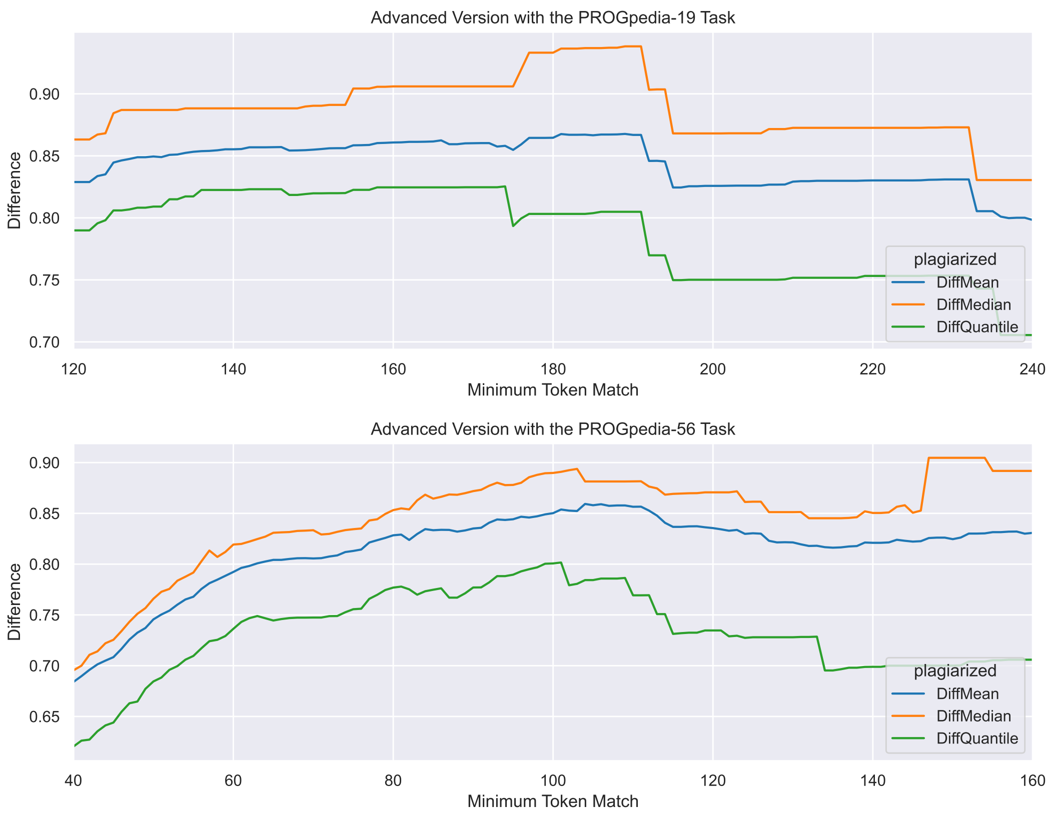

Figure 6.2.: Metrics for different minimum token matches using the advanced language module and the PROGpedia data set

	Programming Homework			PROGpedia	
Min Token Match	**A2016**	**A2017**	**B2016**	**19**	**56**
Dedicated language module	17	17	12	17	6
Simple version	50	45	1200	128	71
Advanced version	71	61	1043	189	101
Optimization level 0	71	61	1043	189	104
Optimization level 1	71	61	1044	154	73
Optimization level 2	71	61	1044	156	74
Optimization level 3	71	61	1044	156	74

Table 6.4.: Best minimum token matches for the different configurations and data sets

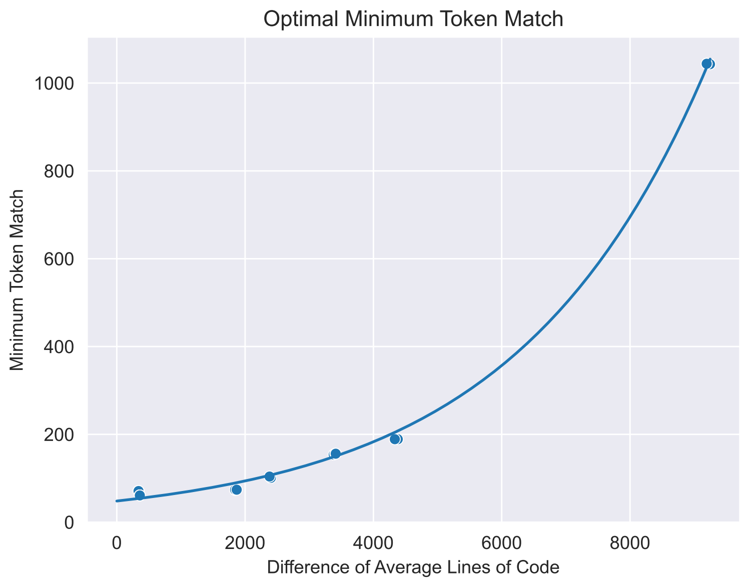

Figure 6.3.: Regression of the minimum token match depending on the difference of average lines of code

However, there is a correlation between the minimum token match and the increase of code lines when compiling the source code. Data sets with a larger increase in code length from the source code to the IR code also have a higher minimum token match. Through a regression of the minimum token matches depending on the difference of average lines of code, as seen in Figure 6.3, we got the following formula to calculate a reference value for other data sets:

$$min\_token\_match(x) = 48.2055162 * e^{0.000333593799*x}$$

$$x = (avg.\ LOC_{LLVM\ IR\ Code}) - (avg.\ LOC_{Source\ Code})$$

With x being the difference between the average lines of code from the LLVM IR code and the average lines of code from the source code.

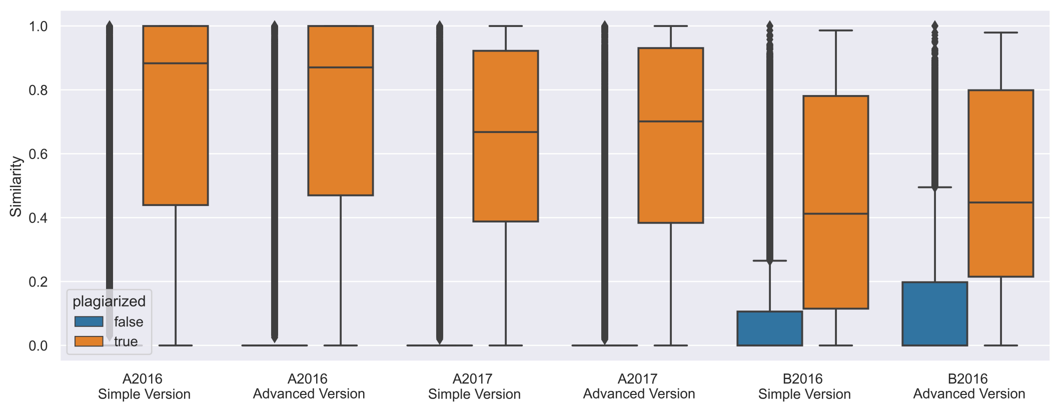

Figure 6.4.: Similarity distributions of plagiarized and non-plagiarized pairs for the simple and the advanced version with the Programming Homework data set

6.2.2. Impact of the Token Abstraction Version

We provide the results of the different token abstraction versions in this section. Furthermore, we derive takeaways for designing new IR language modules based on our findings. The values in bold in our tables are the best compared to the other values.

Programming Homework Data Set

Figure 6.4 shows the similarity distributions of the token abstraction versions for each course of the Programming Homework data set. In the comparison between the simple and the advanced versions, we see that the advanced language module performs slightly better for each course than the simple one. We see that for course A2016, the similarities of the plagiarized pairs of the advanced version are slightly higher than those of the simple version, while the non-plagiarized pairs stay the same. For course A2017, the median of the plagiarized pairs is higher in the advanced version. Lastly, for course B2016, the non-plagiarized pairs have a generally higher similarity in the advanced version, which is undesirable. However, the similarities of the plagiarized pairs are also higher in the advanced version.

Comparing the metrics in Table 6.5, we see the exact results. The table shows the values for the mean, median, and quantile and the difference for each course and version. We see that the metrics are generally higher in the advanced version, which is also reflected in the score. The scores of the simple version are 1.95 for course A2016, 1.60 for A2017, and 0.80 for B2016. The advanced version scores are all higher at 1.97 for course A2016, 1.62 for A2017, and 0.84 for B2016.

So, the advanced version performs better at detecting plagiarized submissions for the Programming Homework data set, even though the differences are small.

Simple Version				
Data Set	Metric	Plagiarized	Non-Plagiarized	Difference
A2016	Mean	0.7028	0.0742	0.6286
	Median	0.8833	0.0	**0.8833**
	Quantile	0.4394	0.0	0.4394
A2017	Mean	0.6173	0.0725	**0.5448**
	Median	0.668	0.0	0.668
	Quantile	0.3884	0.0	**0.3884**
B2016	Mean	0.4544	0.0793	0.3751
	Median	0.4124	0.0	0.4124
	Quantile	0.1152	0.1062	0.009

Advanced Version				
Data Set	Metric	Plagiarized	Non-Plagiarized	Difference
A2016	Mean	0.7131	0.0844	**0.6288**
	Median	0.8707	0.0	0.8707
	Quantile	0.4701	0.0	**0.4701**
A2016	Mean	0.6202	0.0826	0.5376
	Median	0.7015	0.0	**0.7015**
	Quantile	0.3841	0.0	0.3841
A2016	Mean	0.4913	0.1135	**0.3778**
	Median	0.4475	0.0	**0.4475**
	Quantile	0.2155	0.1981	**0.0174**

Table 6.5.: Metrics for the different token abstraction versions of the Programming Homework data set.

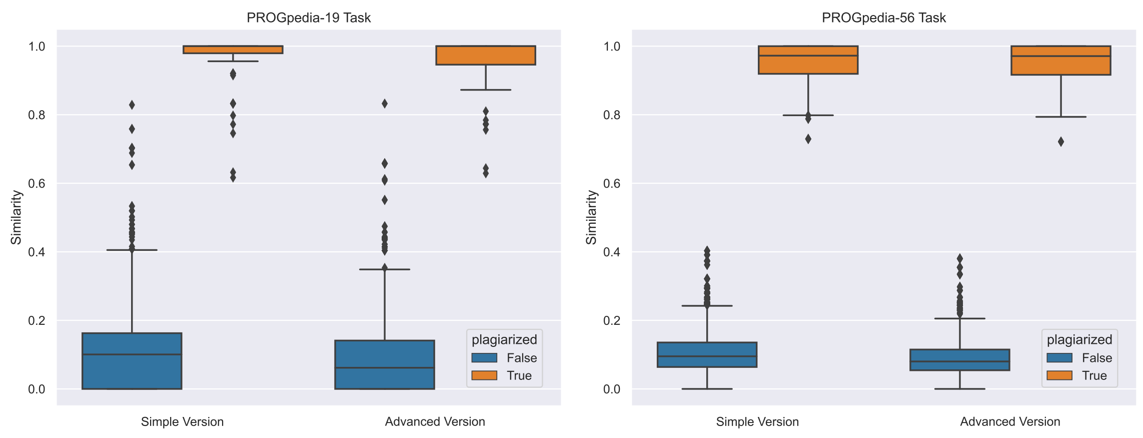

Figure 6.5.: Similarity distributions of plagiarized and non-plagiarized pairs for the simple and the advanced version with the PROGpedia data set

PROGpedia Data Set

Comparing the simple version with the advanced version for the PROGpedia data set in Figure 6.5, we see the following. The advanced version performs slightly better than the simple version for both tasks. The graphic shows the similarity distributions of non-plagiarized and plagiarized pairs for each task. We see that for the PROGpedia-19 task, the similarity of the plagiarized pairs is generally higher for the simple version. However, the similarities of the non-plagiarized pairs are also generally higher. For the PROGpedia-56, task the similarities of the non-plagiarized pairs are lower for the advanced version, while the plagiarized similarities remain the same.

Comparing the metrics in Table 6.6 clarifies the results. We see that the metrics are generally higher for the advanced version, which can also be seen in the score. The score of the PROGpedia-19 task for the simple version is 2.57, while the advanced version performs better, with a higher score of 2.61. The advanced version also performs better for the PROGpedia-56 task, with a score of 2.55 compared to 2.50 for the simple version.

So, the advanced version performs better at detecting plagiarism than the simple version for the PROGpedia data set.

Token Abstraction Strategy

The results from our evaluation show that the advanced version is better at detecting plagiarism than the simple version. This is because the score of the advanced version is higher for both tasks. Therefore, a more extensive token set is more effective, differentiating between different operations and generating tokens for global variables and constants. This indicates the importance of finer-grained types of tokens that capture a larger part of the structure and behavior. Finer-grained token types reduce the overall similarity of

PROGpedia-19 Task						
	Simple Version			**Advanced Version**		
Metric	Plagiarized	Non-Plag.	Diff.	Plagiarized	Non-Plag.	Diff.
Mean	0.9641	0.1107	0.8534	0.9542	0.0865	**0.8677**
Median	1	0.1006	0.8994	1	0.0617	**0.9383**
Quantile	0.9793	0.1626	**0.8167**	0.9461	0.1411	0.8049

PROGpedia-56 Task						
	Simple Version			**Advanced Version**		
Metric	Plagiarized	Non-Plag.	Diff.	Plagiarized	Non-Plag.	Diff.
Mean	0.9406	0.1050	0.8356	0.9395	0.0858	**0.8538**
Median	0.9723	0.0951	0.8772	0.9711	0.0801	**0.8909**
Quantile	0.9193	0.1355	0.7839	0.9165	0.1149	**0.8016**

Table 6.6.: Metrics for the different token abstraction versions of the PROGpedia data set

non-plagiarized pairs while retaining the high similarity of plagiarized pairs. Table A.1 further shows the number of generated tokens and distinct tokens for each version.

Additionally, tokens for basic blocks further lower the similarity of non-plagiarized pairs without affecting the similarity of plagiarized pairs much. This is because we decrease the similarity of programs with different control flows due to generating different tokens. However, common plagiarism and obfuscation techniques, like a verbatim copy and inserting or reordering statements [17] usually do not change the control flow. Therefore, plagiarized pairs are unaffected.

This result indicates that a more extensive token set with finer-grained token types, like differentiating between operations, should be used when creating new IR language modules. The following evaluation will now be carried out on the advanced version as it is proven to be better.

6.2.3. Impact of the LLVM Optimization Level

We provide the results of the LLVM IR language module with different optimization levels. We further discuss the results and make a recommendation for using compiler optimizations. Again, the values in bold in our tables are the best compared to the other values.

Programming Homework Data Set

Figure 6.6 shows the similarity distributions of the Programming Homework data set for each course and optimization level. We see that the distributions are almost indifferent for each optimization level. Therefore, the optimization level does not appear to have an impact on plagiarism detection for the Programming Homework data set.

Data Set	Optimization	$Diff_{Mean}$	$Diff_{Median}$	$Diff_{Quantile}$	Score
A2016	No opt.	0.6288	0.8707	0.4701	1.9695
	Level 0	0.6288	0.8707	0.4701	1.9695
	Level 1	0.6288	0.8707	0.4701	**1.9696**
	Level 2	0.6288	0.8707	0.4701	**1.9696**
	Level 3	0.6288	0.8707	0.4701	**1.9696**
A2017	No opt.	0.5376	0.7015	0.3841	1.6232
	Level 0	0.5376	0.7015	0.3841	1.6232
	Level 1	0.5381	0.7015	0.3841	**1.6237**
	Level 2	0.5381	0.7015	0.3841	**1.6237**
	Level 3	0.5381	0.7015	0.3841	**1.6237**
B2016	No opt.	0.3778	0.4475	0.0174	0.8427
	Level 0	0.3778	0.4475	0.0174	0.8427
	Level 1	0.3796	0.4478	0.0206	**0.848**
	Level 2	0.3796	0.4478	0.0206	**0.848**
	Level 3	0.3796	0.4478	0.0206	**0.848**

Table 6.7.: Metrics of the LLVM IR language module with different optimization levels for the Programming Homework data set

Comparing the results of the data set in Table 6.7, we see a minor improvement in the score from level 0 to level 1. The table shows the metrics for each optimization level of each course. However, apart from that minor improvement between levels 0 and 1, higher optimization levels stay indifferent.

So, a higher optimization level improves the plagiarism detection quality, although the improvement is only negligible.

PROGpedia Data Set

For the Java data sets, we can see the similarity distributions for the optimization levels of the PROGpedia data set in Figure 6.7. We see an improvement from level 0 to level 1 for the PROGpedia-19 task, as the similarities of the non-plagiarized pairs are lower. We can also see an improvement from level 0 to 1 for the PROGpedia-56 task because the similarities of plagiarized pairs are higher, and the non-plagiarized pairs are more compact. Apart from these improvements, higher optimization levels are almost indifferent.

Table 6.8 shows the metrics of each optimization level for the PROGpedia data set. Comparing the scores, we see that optimization level 1 performs the best in both cases. We further notice that the detection quality even decreases slightly with a higher optimization level.

So, using compiler optimization improves plagiarism detection for the PROGpedia data set.

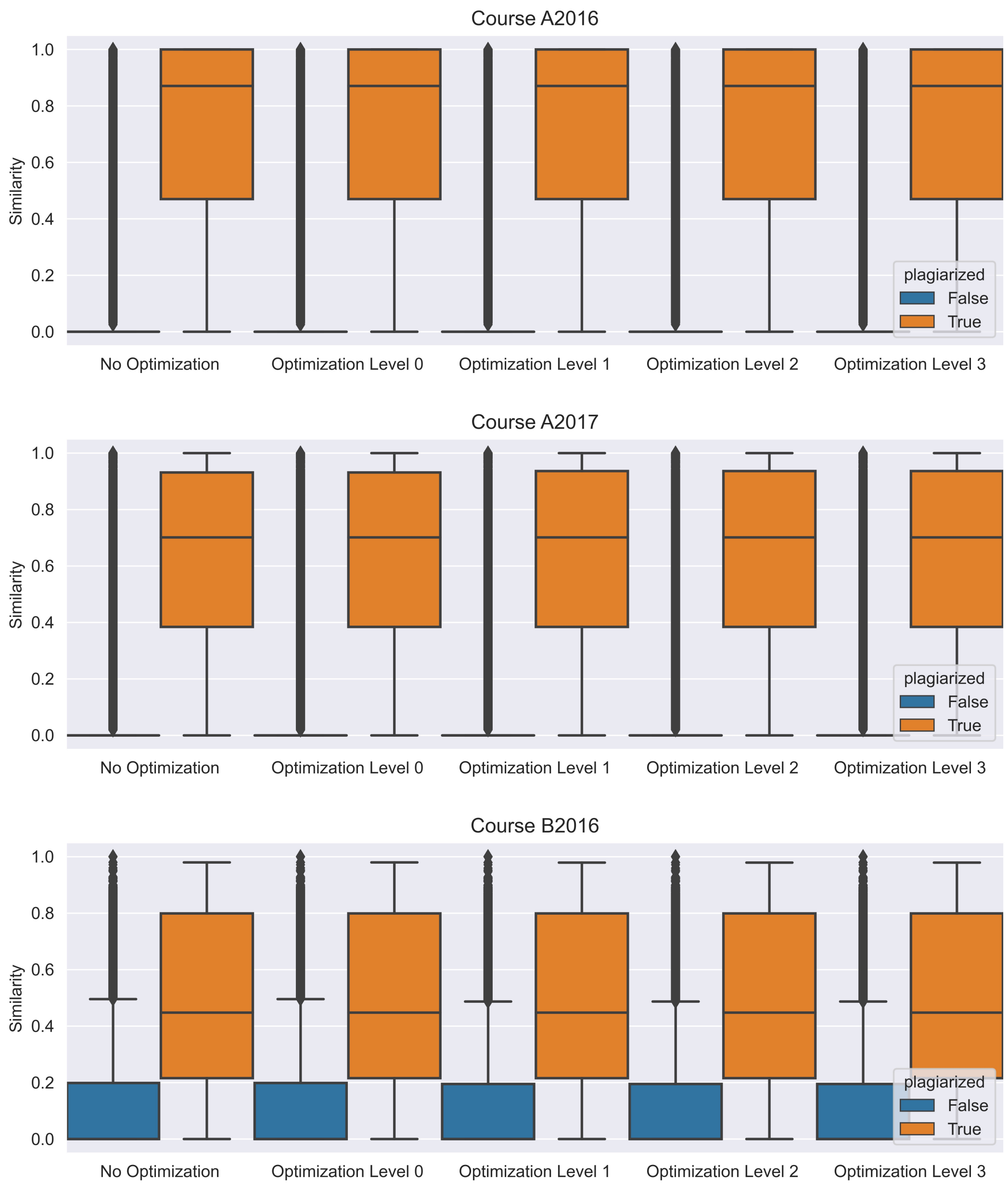

Figure 6.6.: Similarity distributions of plagiarized and non-plagiarized pairs for the different optimization levels with the Programming Homework data set

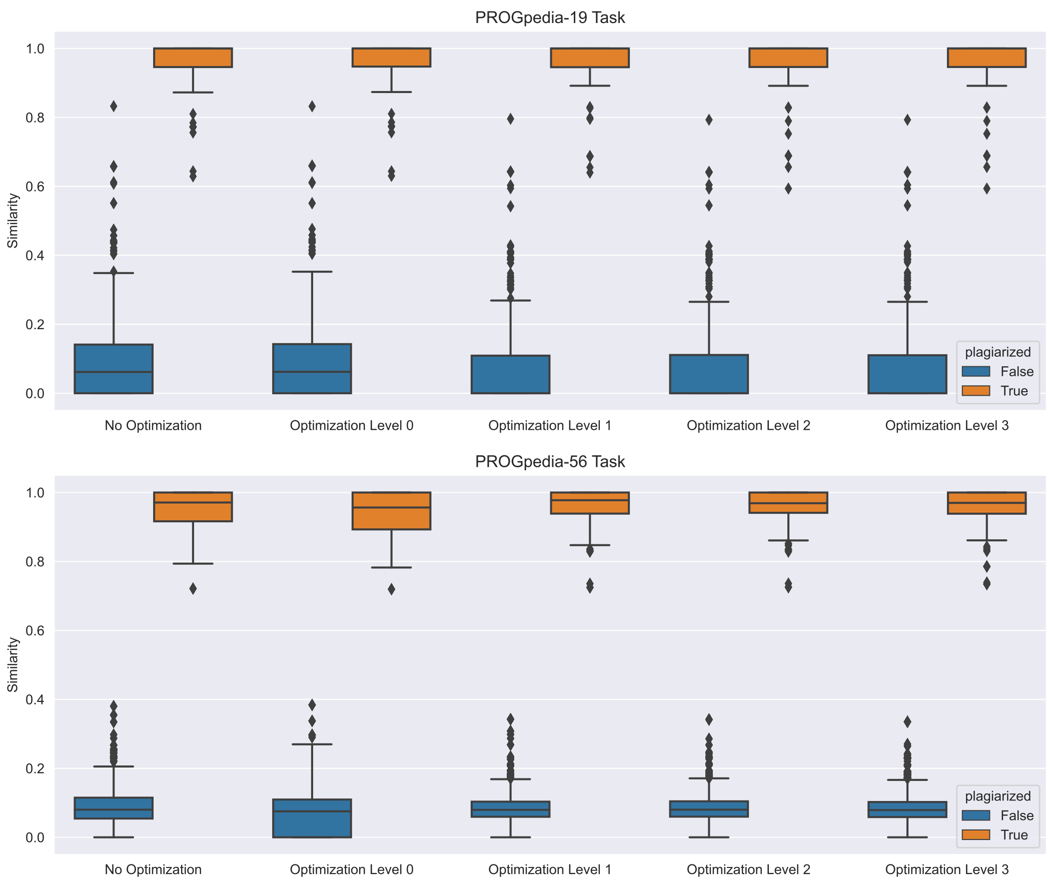

Figure 6.7.: Similarity distributions of plagiarized and non-plagiarized pairs for the different optimization levels with PROGpedia data set

PROGpedia-19 Task				
Optimization	$Diff_{Mean}$	$Diff_{Median}$	$Diff_{Quantile}$	Score
No opt.	0.8677	0.9383	0.8049	2.6108
Level 0	0.8675	0.9378	0.805	2.6104
Level 1	0.8844	1.0	0.8366	**2.7211**
Level 2	0.883	1.0	0.8353	2.7183
Level 3	0.8832	1.0	0.836	2.7192

PROGpedia-56 Task				
Optimization	$Diff_{Mean}$	$Diff_{Median}$	$Diff_{Quantile}$	Score
No opt.	0.8538	0.8909	0.8016	2.5463
Level 0	0.8581	0.8812	0.7834	2.5227
Level 1	0.8657	0.8981	0.8355	**2.5993**
Level 2	0.8636	0.8888	0.8368	2.5892
Level 3	0.8635	0.891	0.8362	2.5907

Table 6.8.: Metrics of the LLVM IR language module with different optimization levels for the PROGpedia data set

Effects of Optimization Levels

Our results clearly show that using compiler optimizations leads to a better plagiarism detection quality. Although the highest optimization level is not necessarily the best.

The LLVM optimizer uses various transformations for the optimization levels. For example, at level 0 all functions marked as *always-inline* are inlined. Level 1 adds optimizations like unrolling loops, deleting dead loops, and branch predictions. Level 2 implements a more extensive inlining technique, and level 3 promotes *by reference* arguments to be *by value* arguments to simplify the code and remove *alloca* instructions.

We assume that plagiarism detection improves because compiler optimizations reduce the variance in an IR code with their optimizing transformations. We can also see in Table 6.2 and Table 6.3 that the optimizations reduce the lines of code. Therefore, the code is more compact, allowing less variance between the IR codes as well. Due to the lower variance, it is easier to detect plagiarized pairs as they are more similar, leading to a better detection quality. Furthermore, this result is similar to what Christian Arwin and S. M. M. Tahaghogi [3] found out. They evaluated plagiarism detection on the GCC RTL and found that higher optimization levels lead to better detection quality. They also justify it by saying that optimizations lead to lower variance, improving plagiarism detection.

We conclude from our results that optimization level 1 provides the best overall results. Therefore, it should be used as a preprocessing technique for our language module to provide the best detection quality. This further indicates that compiler optimizations should also be used for other IR language modules to improve plagiarism detection.

6.2.4. Comparison with Dedicated Language Modules

We show the results of the different language modules and compare them. For the comparison, we use the optimization level 1 for the LLVM IR language module as it delivered the best results. We further discuss the need for dedicated language modules and the advantages of IR language modules. Again, the values in bold in our tables are the best compared to the other values.

Programming Homework Data Set

Figure 6.8 shows the similarity distributions for the dedicated CPP and our LLVM IR language module. We see that the CPP language module performs better for each course, even if it is just slightly. For courses A2016 and A2017, the similarities of the plagiarized pairs are generally higher for the CPP language module, while the non-plagiarized pairs remain the same. For course B2016, the similarities of the non-plagiarized pairs are lower for the CPP language module and the mean of the plagiarized pairs is also higher.

Comparing the metrics in Table 6.9 shows the exact results. The CPP language module performs better for each course, as the metrics are higher than the LLVM IR ones. This

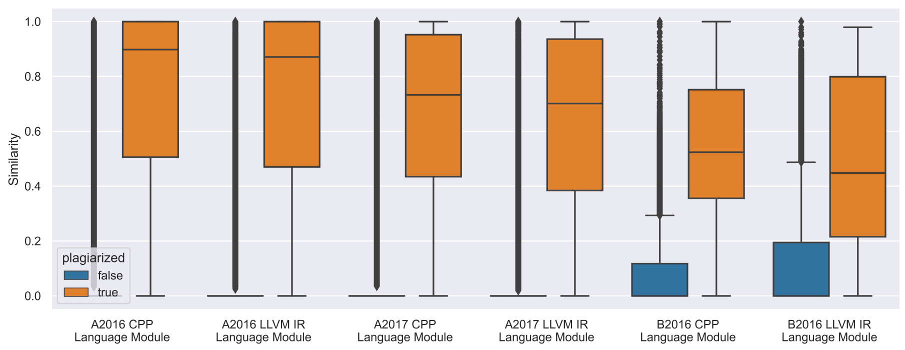

Figure 6.8.: Similarity distributions of plagiarized and non-plagiarized pairs for the CPP and the LLVM IR language module with the Programming Homework data set

can also be seen in the score. The scores for the CPP language module are 2.03 for course A2016, 1.73 for A2017, and 1.22 for B2016. The scores of the IR language module are all lower at 1.97 for course A2016, 1.62 for A2017, and 0.85 for B2016.

This means that the CPP language module performs better at detecting plagiarized submissions for the Programming Homework data set than our LLVM IR language module.

PROGpedia Data Set

Figure 6.9 shows the similarity distributions of the language modules for the PROGpedia data set. We can clearly see that in the case of the PROGpedia-19 task, the Java language module performs better than the IR language module. The similarities of non-plagiarized pairs are lower, and the similarities of plagiarized pairs are higher for the Java language module. In contrast, the IR language module performs significantly better for the PROGpedia-56 task. The similarities of non-plagiarized pairs are lower, while the similarities of the plagiarized pairs are significantly higher for the IR module.

Table 6.10 shows the results of both language modules. Comparing the metrics, we see that the Java language module performs better for the PROGpedia-19 task. For the PROGpedia-56 task, however, the metrics are significantly higher for the IR language module, meaning the IR module performs better. Comparing the scores for the PROGpedia-19 task, the Java language module has a score of 2.95, while the IR language module has a score of only 2.72. Contrarily, regarding the PROGpedia-56 task, the IR language module has a score of 2.60, higher than the score of the Java language module, with 0.73.

So, we clearly see a difference in the detection quality between the two tasks. The Java language module is better at the PROGpedia-19 task than our LLVM IR language module. However, the IR language module shows a significantly better result regarding the PROGpedia-56 task.

CPP Language Module				
Data Set	Metric	Plagiarized	Non-Plagiarized	Difference
A2016	Mean	0.7282	0.0974	**0.6308**
	Median	0.898	0.0	**0.898**
	Quantile	0.5057	0.0	**0.5057**
A2017	Mean	0.6526	0.0922	**0.5604**
	Median	0.7326	0.0	**0.7326**
	Quantile	0.4348	0.0	**0.4348**
B2016	Mean	0.5337	0.0712	**0.4624**
	Median	0.5235	0.0	**0.5235**
	Quantile	0.3558	0.1172	**0.2385**

LLVM IR Language Module				
Data Set	Metric	Plagiarized	Non-Plagiarized	Difference
A2016	Mean	0.7132	0.0844	0.6288
	Median	0.8707	0.0	0.8707
	Quantile	0.4701	0.0	0.4701
A2017	Mean	0.6206	0.0825	0.5381
	Median	0.7015	0.0	0.7015
	Quantile	0.3841	0.0	0.3841
B2016	Mean	0.4911	0.1116	0.3796
	Median	0.4478	0.0	0.4478
	Quantile	0.2154	0.1948	0.0206

Table 6.9.: Metrics of the comparison of the CPP and LLVM IR language module for the Programming Homework data set

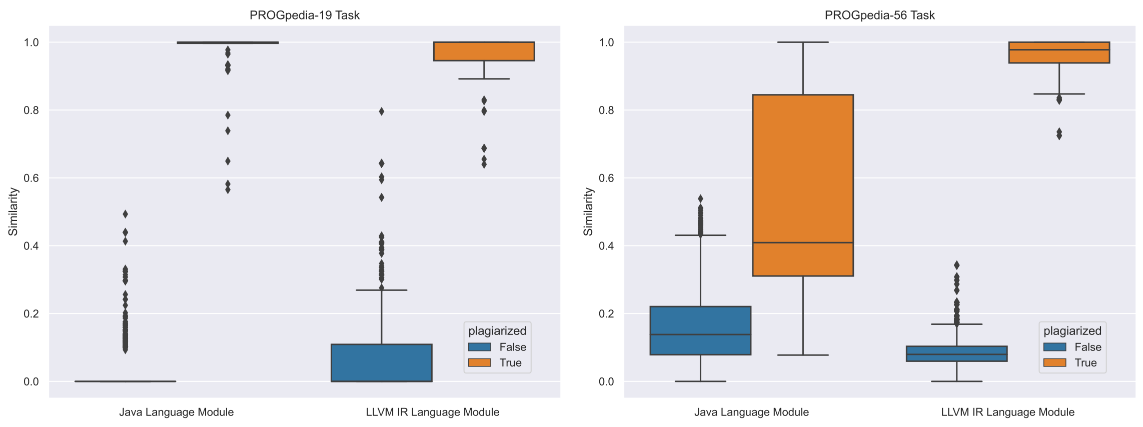

Figure 6.9.: Similarity distributions of plagiarized and non-plagiarized pairs for the Java and the LLVM IR language module with the PROGpedia data set

PROGpedia-19 Task						
	Java Language Module			**LLVM IR Language Module**		
Metric	Plagiarized	Non-Plag.	Diff.	Plagiarized	Non-Plag.	Diff.
Mean	0.9690	0.0202	**0.9488**	0.9483	0.0639	0.8844
Median	1.0	0.0	**1.0**	1.0	0.0	**1.0**
Quantile	0.9967	0.0	**0.9967**	0.9457	0.1090	0.8366

PROGpedia-56 Task						
	Java Language Module			**LLVM IR Language Module**		
Metric	Plagiarized	Non-Plag.	Diff.	Plagiarized	Non-Plag.	Diff.
Mean	0.5211	0.1566	0.3645	0.9496	0.0839	**0.8657**
Median	0.4091	0.1382	0.2709	0.9777	0.0796	**0.8981**
Quantile	0.3107	0.2205	0.0901	0.9388	0.1034	**0.8355**

Table 6.10.: Metrics of the comparison of the Java and LLVM IR language module for the PROGpedia data set

Necessity of Dedicated Language Modules

Our results show that the dedicated language modules perform better at the Programming Homework and the PROGpedia-19 task. Since these data sets consist of student submissions, the dedicated language modules perform better at detecting plagiarism by students than our LLVM IR language module. We further see a higher similarity of non-plagiarized pairs of our LLVM IR language module for the PROGpedia-19 task and the B2016 course, meaning the IR language module also has a higher false-positive rate than the dedicated modules. Therefore, we conclude that the additional abstraction level of compiling the source code to an IR results in a loss of information relevant to detecting plagiarism. However, dedicated language modules do not perform much better than our IR language module. Still, the results suggest that it is preferred to have specific language modules for each programming language to improve plagiarism detection in student submissions. This indicates that dedicated language modules are necessary for the best detection quality, even if IR language modules support multiple programming languages.

However, the IR language module shows significantly better resilience to obfuscation attacks than the dedicated Java language module. We can see that from the better detection quality of the LLVM IR module with the PROGpedia-56 task, which consists of automatically obfuscated source codes. This resilience comes from the compilation step, not from optimizations because the difference between no compiler optimizations and optimization level 1 is relatively small, as Figure 6.7 shows. LLVM already uses various optimizations by default when compiling the code. The plagiarism was generated by adding new statements and reordering existing statements. The dead code elimination performed at the compilation step already removes most of the new statements, making it more resilient to this attack. Furthermore, we assume that with the code transformed into SSA form, the language module also becomes more resilient to reordering statements. On the other hand, the Java language module does not use such techniques, making it more vulnerable. This result is also consistent with Devore-McDonald and Berger [8], who suspect that plagiarism detection on binary or low-level languages is more resilient to obfuscation attacks. This also explains the difference in the results of the PROGpedia data set, as the PROGpedia-19 task contains student submissions, and the PROGpedia-56 task automatically obfuscated code.

As students can use frameworks to obfuscate their plagiarized code automatically [8], an IR language module can help detect such plagiarism. With this, and with support for multiple languages and performance not significantly worse than dedicated language modules, a language module for an IR can still be useful.

6.3. Limitations

There are several limitations to our language module and our evaluation.

An inherent limit of intermediate representations is that they are architecture-dependent. Therefore, the same source code will produce different IR codes when compiled on different architectures. This will ultimately result in a lower similarity as different tokens get generated. Instructors would then get different results if they use different architectures or if students compile the code themselves. We prevented this for our evaluation by compiling all source codes on the same architecture to the LLVM IR. A solution for student submissions is that one or all instructors perform the compilation on the same architecture instead of each student separately.

Another limitation is that our language module needs LLVM IR code as an input. Therefore, for student submissions, an instructor must compile each submission beforehand. This process is time-consuming, especially if the compilation is not straightforward, like in the case of Java. This step is unnecessary for dedicated language modules, which saves time. However, this does not affect the results of our evaluation.

JPlag allows the user to view the pairs of submissions and marks all the parts of the code that got matched by the algorithm. This allows an instructor to compare the submissions more easily and decide if it is an actual case of plagiarism. For our LLVM IR language module, JPlag only shows the matches between the IR codes and not between the actual source codes. Therefore, comparing the submission with the dedicated language module is easier as it shows the matches between the source code. As this affects the usability of the language module in JPlag, it is irrelevant to the results of our evaluation.

6.4. Threats to Validity

Several factors limit the validity of our evaluation.

The choice of token types and token extraction may not be optimal for the IR language module because there could be a better set of token types and token extraction rules that score better results. However, this is unlikely as we put great effort into creating our language module by testing different token abstraction strategies. Another effect is the actual occurrence of token types, as there are token types with zero or only a few occurrences. This is because several language features, like inline assembly, are absent in the data sets we used. Therefore, our results might differ from results on other data sets.

The metrics we used for our evaluation may not represent the quality of a language module. While usually, a more significant difference between non-plagiarized and plagiarized pairs is more desirable, there might be metrics that represent this better. As the metrics play an integral role in comparing the configurations and language modules, different metrics might return different results.

As described in subsubsection 6.1.3.4, we made several modifications to the PROGpedia data set to lower the runtime and increase plagiarism detection quality. We can not guarantee that no relevant information gets lost with these modifications, but since this was done for every file, this applies to every file the same. Furthermore, we rely on our advisors's judgment of plagiarism for the ground truth, supported by JPlag's Java language module.

A possible thread for the Programming Homework data set is that it contains many pairs which are not labeled as plagiarized but score a very high similarity or even are identical. This negatively affects the similarities of non-plagiarized pairs and tends to result in a higher false-positive rate, but applies to the IR language module and the CPP language module the same.

Lastly, the results of our evaluation are limited to the data sets, with their characteristics of complexity, submission length, and used language features. The source codes from the used data sets are real-world assignments but only a few hundred lines. In contrast, there are assignments with more than a thousand lines. Furthermore, we only evaluate Java and C/C++ as source languages, while more languages can be used as multiple languages can be compiled to LLVM IR.

In order to improve the reproducibility of our evaluation results, we published our implementation of the language module together with the test data and evaluation scripts [11].

7. Future Work

Our work focused on comparing dedicated language modules with an IR language module regarding plagiarism detection. We created a language module for the LLVM IR and compared it to existing dedicated language modules for our evaluation.

Future work might continue evaluating the language module with different data sets. The average number of LOC of the programs in our data sets is only a few hundred. In contrast, there are real-world assignments that have more than a thousand lines. Furthermore, data sets with different complexity and used language features could also be evaluated to see if the results are consistent.

We only used Java and C/C++ as source languages for our evaluation, but as mentioned, the LLVM IR supports various programming languages. Therefore, future work could compare our LLVM IR language module with dedicated language modules of other languages, like Rust or Python. This would provide a more accurate statement about the need for dedicated language modules. However, the limiting factor for such languages is the lack of data sets that already contain a ground truth.

Another possible work could address the differences between different architectures. As mentioned before, the IR is architecture-dependent, and different tokens are generated on different architectures. Therefore, one could evaluate the differences between different architectures for the same data set.

Future work might also address other intermediate representations. An IR like Java Bytecode might provide better results in detecting plagiarism as it supports more homogeneous languages based on the Java virtual machine. Therefore, the compilation step might lose less relevant information, providing better results. With an IR similar to the LLVM IR, such as the GCC IR, it could be shown if the results are consistent between different intermediate representations or if there are significant differences.

Our evaluation shows that the IR language module is more resilient to obfuscation attacks than the dedicated Java language module. As a next step, it could be evaluated how well it copes with different kinds of obfuscation attacks, for example, generated by MOSSAD [8]. Future work might assess what exactly results in resilience against obfuscation attacks and could develop methods to improve plagiarism detection based on these findings. These methods could be used to improve dedicated language modules to make them more resilient towards such obfuscation attacks.

8. Conclusion

Token-based plagiarism detection is a state-of-the-art technique for detecting plagiarism in programming assignments. However, supporting a new programming language usually involves the creation of a new language module for that particular language on the assumption that this improves plagiarism detection. This process is very time-consuming, and it is unclear whether it actually improves plagiarism detection. In this thesis, we evaluated the need for dedicated language modules by comparing them to an IR language module supporting multiple programming languages. Furthermore, we derived takeaways for designing and implementing new language modules for IRs. We created a language module for the LLVM IR by describing the mapping of the LLVM IR to tokens and explaining our design ideas. For the evaluation, we used real-world data sets of student submissions and compared our language module with the Java and C/C++ language modules in JPlag.

The evaluation highlights several factors that influence the ability of a token-based plagiarism detector to detect possible cases of plagiarism. These include the minimum token match parameter, the optimization level, and the token extraction strategy. Regarding designing IR modules, we show that a higher minimum token match is necessary as one source code line gets mapped to several IR code lines. We furthermore provide a formula to calculate a reference value for the minimum token match to ease usability. Higher optimization levels lead to better detection quality as compiler optimizations lower the variance between plagiarized pairs, making detecting them easier. Consequently, we recommend the use of compiler optimizations to improve plagiarism detection. Our advanced version with more and finer-grained token types proved more effective than the simple version regarding the design and token abstraction strategy. Therefore, a more extensive token set is more effective, differentiating between different operations and generating tokens for global variables and constants. This indicates the importance of finer-grained tokens that capture a more significant part of the structure and behavior, resulting in a lower similarity of non-plagiarized pairs.

Regarding the necessity of dedicated language modules, our results show that the two dedicated language modules for C/C++ and Java perform better at detecting plagiarism by students than our LLVM IR language module. We assume that the additional abstraction level of compiling the source code to IR results in a loss of information relevant to detecting plagiarism. Therefore, the results suggest that dedicated language modules for a specific language are better than IR language modules supporting multiple languages at detecting plagiarism. This indicates that dedicated language modules are necessary, to gain the best detection quality and it is preferred to have a language module for each programming language to improve plagiarism detection.

However, the IR language module shows significantly better resilience to obfuscation attacks, like inserting new statements or reordering existing ones, than the dedicated Java language module. As students can use frameworks like MOSSAD [8] to obfuscate their plagiarized code automatically, it will be easier than ever to attack plagiarism detectors. While our IR language module performs worse than dedicated language modules today, with the use of such frameworks, this may change in the future. Then, an IR language module showing certain resilience can help detect such plagiarism. With this, and with support for multiple languages and performance not significantly worse than dedicated language modules, a language module for an IR can still be useful.

Bibliography

[1] Bowen Alpern, Mark N Wegman, and F Kenneth Zadeck. “Detecting equality of variables in programs”. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1988, pp. 1–11.

[2] *ANTLR Grammars*. URL: `https://github.com/antlr/grammars-v4/tree/master/llvm-ir` (visited on 06/10/2023).

[3] Christian Arwin and S. M. M. Tahaghoghi. “Plagiarism Detection across Programming Languages”. In: *Proceedings of the 29th Australasian Computer Science Conference - Volume 48*. ACSC ’06. Hobart, Australia: Australian Computer Society, Inc., 2006, pp. 277–286. ISBN: 1920682309.

[4] Max Brunsfeld et al. *tree-sitter/tree-sitter: v0.20.0*. Version v0.20.0. June 2021. DOI: `10.5281/zenodo.5044536`. URL: `https://doi.org/10.5281/zenodo.5044536`.

[5] Pedro M. Caldeira et al. “Improving Syntactical Clone Detection Methods through the Use of an Intermediate Representation”. In: *2020 IEEE 14th International Workshop on Software Clones (IWSC)*. 2020, pp. 8–14. DOI: `10.1109/IWSC50091.2020.9047637`.

[6] Daniela Chuda et al. “The Issue of (Software) Plagiarism: A Student View”. In: *IEEE Transactions on Education* 55.1 (2012), pp. 22–28. DOI: `10.1109/TE.2011.2112768`.

[7] Georgina Cosma and Mike Joy. “Towards a Definition of Source-Code Plagiarism”. In: *IEEE Transactions on Education* 51.2 (2008), pp. 195–200. DOI: `10.1109/TE.2007.906776`.

[8] Breanna Devore-McDonald and Emery D. Berger. “Mossad: Defeating Software Plagiarism Detection”. In: *Proc. ACM Program. Lang.* 4.OOPSLA (Nov. 2020). DOI: `10.1145/3428206`. URL: `https://doi.org/10.1145/3428206`.

[9] *GraalVM*. URL: `https://www.graalvm.org` (visited on 06/10/2023).

[10] Hannes Greule. “Evidence-based Token Abstraction for Software Plagiarism Detection”. Bachelor’s Thesis. Karlsruhe Institute of Technology, 2023.

[11] Niklas Rainer Heneka. *Software Plagiarism Detection on Intermediate Representation Data Set*. Version 1.0.0. Zenodo, Oct. 2023. DOI: `10.5281/zenodo.8403147`. URL: `https://doi.org/10.5281/zenodo.8403147`.

[12] Petri Ihantola et al. “Review of Recent Systems for Automatic Assessment of Programming Assignments”. In: *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*. Koli Calling ’10. Koli, Finland: Association for Computing Machinery, 2010, pp. 86–93. ISBN: 9781450305204. DOI: `10.1145/1930464.1930480`. URL: `https://doi.org/10.1145/1930464.1930480`.

[13] Yoon-Chan Jhi et al. "Program Characterization Using Runtime Values and Its Application to Software Plagiarism Detection". In: *IEEE Transactions on Software Engineering* 41.9 (2015), pp. 925–943. DOI: `10.1109/TSE.2015.2418777`.

[14] Yoon-Chan Jhi et al. "Value-Based Program Characterization and Its Application to Software Plagiarism Detection". In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE '11. Waikiki, Honolulu, HI, USA: Association for Computing Machinery, 2011, pp. 756–765. ISBN: 9781450304450. DOI: `10.1145/1985793.1985899`. URL: `https://doi.org/10.1145/1985793.1985899`.

[15] M. Joy and M. Luck. "Plagiarism in programming assignments". In: *IEEE Transactions on Education* 42.2 (1999), pp. 129–133. DOI: `10.1109/13.762946`.

[16] JPlag. *JPlag GitHub Repository*. URL: `https://github.com/jplag/JPlag` (visited on 05/24/2023).

[17] Oscar Karnalim. "Detecting source code plagiarism on introductory programming course assignments using a bytecode approach". In: *2016 International Conference on Information & Communication Technology and Systems (ICTS)*. 2016, pp. 63–68. DOI: `10.1109/ICTS.2016.7910274`.

[18] Cynthia Kustanto and Inggriani Liem. "Automatic Source Code Plagiarism Detection". In: *2009 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing*. 2009, pp. 481–486. DOI: `10.1109/SNPD.2009.62`.

[19] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. 2004, pp. 75–86. DOI: `10.1109/CGO.2004.1281665`.

[20] Chris Lattner and Vikram Adve. "The LLVM Compiler Framework and Infrastructure Tutorial". In: *Languages and Compilers for High Performance Computing*. Ed. by Rudolf Eigenmann, Zhiyuan Li, and Samuel P. Midkiff. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 15–16. ISBN: 978-3-540-31813-2. DOI: `https://doi.org/10.1007/11532378_2`.

[21] Chris Lattner and Vikram Adve. "The LLVM instruction set and compilation strategy". In: *CS Dept., Univ. of Illinois at Urbana-Champaign, Tech. Report UIUCDCS* (2002).

[22] Vedran Ljubovic. *Programming Homework Dataset for Plagiarism Detection*. 2020. DOI: `10.21227/71fw-ss32`. URL: `https://dx.doi.org/10.21227/71fw-ss32`.

[23] *LLVM Project*. URL: `https://llvm.org` (visited on 05/25/2023).

[24] Rien Maertens et al. "Dolos: Language-agnostic plagiarism detection in source code". In: *Journal of Computer Assisted Learning* 38.4 (2022), pp. 1046–1061. DOI: `https://doi.org/10.1111/jcal.12662`.

[25] Matija Novak, Mike Joy, and Dragutin Kermek. "Source-Code Similarity Detection and Detection Tools Used in Academia: A Systematic Review". In: *ACM Trans. Comput. Educ.* 19.3 (May 2019). DOI: `10.1145/3313290`. URL: `https://doi.org/10.1145/3313290` (visited on 05/24/2023).

[26] José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. "PROGpedia: Collection of source-code submitted to introductory programming assignments". In: *Data in Brief* 46 (2023), p. 108887. ISSN: 2352-3409. DOI: https://doi.org/10.1016/j.dib.2023.108887. URL: https://www.sciencedirect.com/science/article/pii/S2352340923000057.

[27] Alan Parker and James O Hamblen. "Computer algorithms for plagiarism detection". In: *IEEE Transactions on Education* 32.2 (1989), pp. 94–99. DOI: 10.1109/13.28038.

[28] Terence J. Parr and Russell W. Quong. "ANTLR: A predicated-LL(k) parser generator". In: *Software: Practice and Experience* 25.7 (1995), pp. 789–810. DOI: https://doi.org/10.1002/spe.4380250705. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380250705. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380250705.

[29] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. "Finding plagiarisms among a set of programs with JPlag". In: *Journal of Universal Computer Science* 8.11 (2002), pp. 1016–1038. URL: https://ps.ipd.kit.edu/downloads/za_2002_finding_plagiarisms_jplag.pdf (visited on 05/22/2023).

[30] LLVM project. *Clang: a C language family frontend for LLVM*. URL: https://clang.llvm.org (visited on 05/27/2023).

[31] Faqih Salban Rabbani and Oscar Karnalim. "Detecting source code plagiarism on. NET programming languages using low-level representation and adaptive local alignment". In: *Journal of Information and Organizational Sciences* 41.1 (2017), pp. 105–123. DOI: https://doi.org/10.31341/jios.41.1.7.

[32] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. "Winnowing: Local Algorithms for Document Fingerprinting". In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. SIGMOD '03. San Diego, California: Association for Computing Machinery, 2003, pp. 76–85. ISBN: 158113634X. DOI: 10.1145/872757.872770. URL: https://doi.org/10.1145/872757.872770 (visited on 05/24/2023).

[33] Farhan Ullah et al. "Software plagiarism detection in multiprogramming languages using machine learning approach". In: *Concurrency and Computation: Practice and Experience* 33.4 (2021). e5000 cpe.5000, e5000. DOI: https://doi.org/10.1002/cpe.5000. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.5000. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5000.

[34] Michael J Wise. "String similarity via greedy string tiling and running Karp-Rabin matching". In: *Online Preprint, Dec* 119.1 (1993), pp. 1–17.

[35] Jitendra Yasaswi et al. "Unsupervised Learning Based Approach for Plagiarism Detection in Programming Assignments". In: *Proceedings of the 10th Innovations in Software Engineering Conference*. ISEC '17. Jaipur, India: Association for Computing Machinery, 2017, pp. 117–121. ISBN: 9781450348560. DOI: 10.1145/3021460.3021473. URL: https://doi-org.ezproxy-kit-1.redi-bw.de/10.1145/3021460.3021473.

A. Appendix

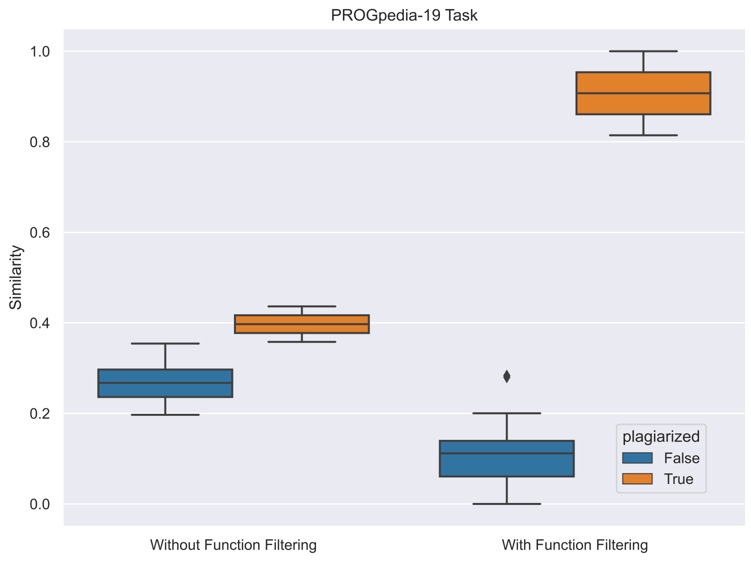

Figure A.1.: Similarity distributions without and with function filtering on a 10% subset of the PROGpedia-19 task with base code

Average Number of Generated Tokens		
Data Set	Simple Version	Advanced Version
PROGpedia-19	3130	4127
PROGpedia-56	1752	2304
Programming Homework A2016	287	397
Programming Homework A2017	289	416
Programming Homework B2016	7570	9652

Average Number of Distinct Tokens		
Data Set	Simple Version	Advanced Version
PROGpedia-19	18	30
PROGpedia-56	17	29
Programming Homework A2016	15	22
Programming Homework A2017	15	22
Programming Homework B2016	21	38

Table A.1.: Average number of generated tokens for each version of the LLVM IR language module

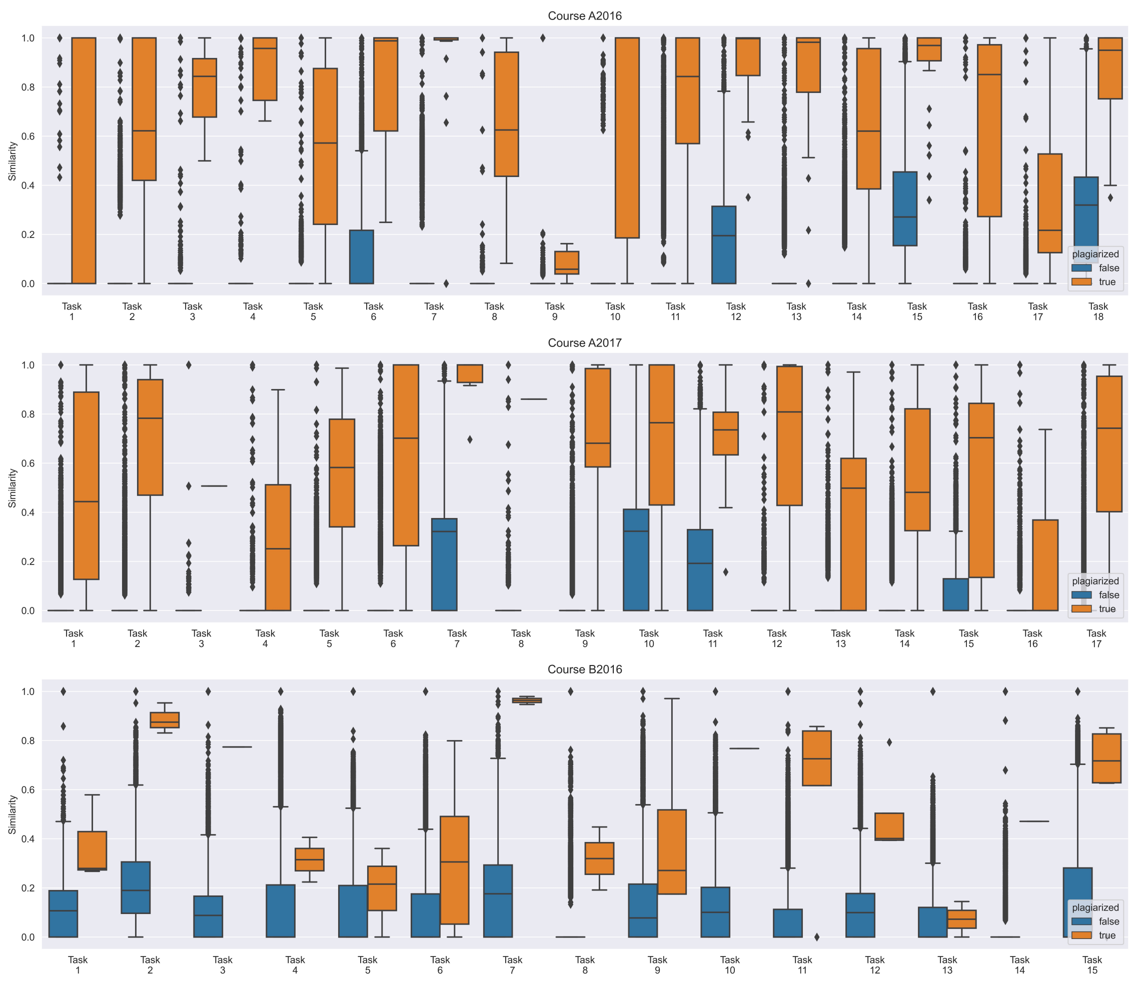

Figure A.2.: Similarity distributions of the different tasks for each course from the Programming Homework data set with the LLVM IR language module