Bachelor's Thesis

USING CHARACTER-BASED GIT BLAME INFORMATION TO ENHANCE THE PRECISION OF COMMIT-INTERACTION ANALYSIS

LEONIE VON MANN EDLE VON TIECHLER

March 28, 2024

Advisor:
Sebastian Böhm Chair of Software Engineering
Florian Sattler Chair of Software Engineering

Examiners:
Prof. Dr. Sven Apel Chair of Software Engineering
Prof. Dr. Sebastian Hack Compiler Design Lab

Chair of Software Engineering
Saarland Informatics Campus
Saarland University

UNIVERSITÄT DES SAARLANDES

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,_____ _____

(Datum/Date) (Unterschrift/Signature)

ABSTRACT

The version control system Git is a popular tool used by many software developers to manage their software projects. It stores project histories and helps with effectively cooperating on a project by providing branching and by working as a distributed system. Combining software analysis with repository information can be helpful for understanding how software projects evolve over time. The version control system Git offers the `blame` functionality to connect repository information with source code. It does so by assigning to each line of code the hash of the commit that last modified it. However, it often occurs that only parts of a line get modified. In these cases the line-based `blame` information restricts us from identifying which authors modified which parts of a line because the last modification is attributed to the whole line. We aim to improve the precision of software analysis involving repository information by computing git `blame` information on a per-character basis.

Software analyses often work on an intermediate representation (IR) that is generated from a program's AST. Therefore, we introduce an AST-based `blame` algorithm that enhances the precision of commit annotations by computing a separate `blame` for each AST node. The main idea is to follow the history of the code of an AST node back until the point where it was first introduced. We use the commit hash that is returned by the line-based git `blame` as a basis and determine whether there has been an actual change at the given position. If there is no change we continue searching for the actual change starting from the previous commit. If there is a change, we take the commit returned by the line-based `blame`.

We evaluate the precision of our improved AST-based `blame` algorithm by comparing the results to a manually derived baseline and the line-based `blame` information. In our experiments, we observe a significant improvement over the line-based `blame` information. To demonstrate how the more precise AST-based `blame` information can impact program analysis, we investigate its effects on the commit-interaction analysis from Sattler et al. [10], an integrated approach that combines low-level program analysis with high-level repository information. This way, we obtain insights on what we can gain from more fine-grained information. In the case of a central-code analysis, we are able to more precisely identify commits that are central in the data-dependency structure of a program.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

The version control system Git[1] is a popular tool used by many software developers to manage their software projects. The Git repository hosting service *GitHub* reports over 100 million users[2] and a survey of *Stack Overflow* states that 93% of software developers use Git[3]. Git stores a projects history and helps with effectively cooperating on a project.

The project history consists of multiple commits describing the changes made to the source code files. Especially in projects that are worked on by multiple developers, it is crucial that each commit is assigned an author to allow the team to identify who made specific changes. With this assignment we know who introduced certain code segments because each commit also has information about the changes made to the source code. For example, this can help with legal issues on code ownership and improve the work flow of software engineers. Another benefit of the information about the author is that people working on the project know who to contact with questions regarding a specific code segment. Besides the issue of determining authorship, repository information, such as which commit introduced what code and when, can be helpful in understanding how software projects evolve over time.

A Git repository stores this information about the authors as metadata. This underlying data and metadata is essential for history tracking and it provides analysis tools with the information necessary for their analysis. The research area of repository mining deals with analysing the vast amount of data from software repositories and using the results to better understand and to improve the process behind developing software [12]. For example, Sattler et al. [10] use repository information coupled with a data-flow analysis to gain a better insight into connections between software developers. We investigate analyses surrounding commit interactions, i. e., examining relationships and interactions between different commits. However, there are many different analyses that use `blame` information. One other example is the identification of commits that introduced or solved bugs [5, 6].

Git provides the `blame` functionality to help with this code attribution. For each line of code, `git blame` tells us the last commit that changed it and who authored that change. However, the problem with this line-based approach is that it only gives us information about the last time someone changed a line, regardless of how large or small the change might have been. Sometimes, commits change only parts of a line of code. For example, they only change one parameter of a function call or they rename a variable. In these cases, the authorship of the full line now gets assigned to the one making the small adjustments. That way, information about what commit first introduced the functionality and perhaps provided a significant amount of code to the project can be hidden by such small changes.

1 https://git-scm.com/
2 https://github.blog/2023-01-25-100-million-developers-and-counting/
3 https://stackoverflow.blog/2023/01/09/beyond-git-the-other-version-control-systems-developers-use/

1.1 GOAL OF THIS THESIS

The goal of this thesis is to introduce a more fine-grained `blame` functionality, that is able to overcome the line-based attribution problem. Our fine-grained `blame` assigns commit annotations to LLVM[4] *intermediate representation (IR)* instructions via a modified CLANG[5] compiler.

As an example, in Figure 1.1, we see commit `db68f6` modifying one line of code and the corresponding change in the *IR* of that line. The modification of the line only adds one *IR* instruction that represents the subtraction of 1. With the line-based `blame`, all four instructions corresponding to the line are mapped to `db68f6`, since it was the last commit to modify that line. However, if we consider the actual change that the commit introduces, it is more correct to only map the instruction for the subtraction of 1 to commit `db68f6` and leave the rest of the instructions mapped to a previous commit.

```
 int foo(int x) {
     int result;
-    result = x + 42;
+    result = x + 42 - 1;
     return result;
 }
```

```
     %1 = load i32, i32 %x.addr
     %add = add nsw i32 %1, 42
+    %sub = sub nsw i32 %add, 1
     store i32 %sub, i32* %result
```

Figure 1.1: Change in source code from commit `db68f6`

In this thesis, we introduce an algorithm that calculates such a fine-grained mapping from commit information to *IR* instructions. Our implementation utilizes CLANG's abstract syntax tree (AST) to determine the characters belonging to each AST node. A character-based `blame` algorithm then computes the correct commit hash for the instructions belonging to the AST node. To evaluate our optimized `blame` computation, we compare its results to the line-based `blame` on multiple projects. Additionally, we show the effects it has on the integrated analysis approach introduced by Sattler et al. [10]. More specifically, we inspect the results of a commit-interaction analysis dealing with the identification of commits affecting central code.

1.2 OVERVIEW

In Chapter 2, we provide background information about GIT, diff algorithms, CLANG, and VARA. Afterwards, we look at how we implement the optimized `blame` computation in Chapter 3. The implementation chapter contains the compiler integration, introduces a character-based `blame` algorithm, and discusses limitations. Next, we present the research questions and the test repositories with which we evaluate them in Chapter 4. In the following Chapter 5, we evaluate the results we obtain in our experiments. At first, we discuss the benefits of more fine-grained information on common code development scenarios. Then, we look at the quantitative difference to the previous line-based blame. Finally, we investigate the impact it has on commit-interaction analysis by running a central-code

4 https://llvm.org/
5 https://clang.llvm.org/

analysis on program files annotated with AST-based `blame` and comparing its results to the line-based ones. The next Chapter 6 presents prior research in authorship attribution and in exploring code development. In the last Chapter 7, we conclude and provide an outlook on possible future work.

2 BACKGROUND

To understand how we intend to improve the current `git blame`, first, we need to understand specific details about Git: how it operates and what information `git blame` provides, the various diff algorithms it uses, the adapted diffing we also use later on in the implementation. Then, we briefly introduce the LibGit2 library[1]. Another important aspect is the process into which we incorporate our improved commit mapping algorithm. As we realize the commit annotations in a modified Clang compiler, used by an anlaysis framework called VaRA[2], we introduce these tools at the end of the chapter.

2.1 GIT

Git is a version control system that persists the development history of a project in a repository. A repository is made up of a database of blobs, a set of commits, and metadata. A commit can be seen as a snapshot of the project at a certain point in time containing metadata, a list of ancestor commits, and referring to blobs which contain the project data [4]. Commits are identified by commit hashes. The metadata of a commit provides information about the time of the commit, the author, and the commit message.

Git provides the `blame` functionality, which maps each line of code in the repository to the commit hash that last modified the line. It can be used as a command line tool[3] but is also often integrated in different GUIs. The `git blame` tool annotates each line in the selected files with a timestamp, the commit which last modified it, and by that, the author who did the modification. Figure 2.1 gives an example for a file with its corresponding `blame` results. We can see that the code in Line 3 was last modified by commit `db68f6` while the remaining lines were last modified by commit `ad1a64`. However, we cannot say whether Line 3 was first introduced by that commit or if there was only parts of the line that were changed or added. This is exactly the information we aim to make more precise in this thesis.

```
1    int foo(int x) {                    ↦ ad1a64
2        int result;                     ↦ ad1a64
3        result = x + 42 - 1;            ↦ db68f6
4        return result;                  ↦ ad1a64
5    }                                   ↦ ad1a64
```

Figure 2.1: Example git `blame` annotation

1 https://libgit2.org/
2 https://vara.readthedocs.io/en/vara-dev/research_tool_docs/vara/vara.html
3 https://git-scm.com/docs/git-blame

2.2 LIBGIT2

LibGit2 is a software library that provides most Git core methods to use in many different programming languages. We use it to interact with Git repositories programmatically in C++. The functionalities that LibGit2 provides range from modifying the repository by, e. g., creating commits and branches to retrieving repository information like the line-based `blame` and using the `diff` functionality. For this thesis, the important part is the extracting of information. With LibGit2, we can obtain the necessary information to compute the character-based `blame`. Specifically, we use functionality that retrieves the line-based `blame`, parent commits, diff hunks, and the content of the project corresponding to a commit.

2.3 DIFF ALGORITHMS

Diff algorithms are algorithms that, given two strings of text, compute their difference. This means they return information about the location of changed, removed, or newly introduced parts. The output of such algorithms usually is a sequence of insertions and deletions, which describes how to transform the old text to the new one. In Git, these changes to text sequences are typically referred to as hunks. In the context of `git blame`, diff algorithms are used to compare lines and figure out when a line was introduced or last changed so that the correct commit can be assigned. This shows that the `blame` functionality heavily relies on diff algorithms.

Git offers four diff algorithms: Myers, Minimal, Patience, and Histogram[4]. The default algorithm used by `git diff` is the Myers algorithm [8]. Nugroho et al. [9] conclude that out of the four diff algorithms the Histogram provides better results than the Myers algorithm if applied on a line basis. They state that, since the Myers algorithm only detects identical lines, the identified Git hunks are not always as precise as the ones returned from Histogram. However, in the implementation presented in this thesis, we compare the individual lines on a character basis, so in our case, there is no advantage in using the Histogram algorithm, which is why we use the Myers algorithm.

The Myers algorithm transforms the two text sequences into a graph representation and subsequently converts the task of computing the difference of these sequences into a shortest path problem. Given two character sequences A with length N and B with length M, the original algorithm computes a minimal edit script, a sequence of operations that provides information about how to transform A into B. To do so, a $N \times M$ grid is created. Each intersection of the grid represents a node (x, y) and each node is connected horizontally and vertically to its neighbours. The variables x and y are the character indices in A and B respectively. Each horizontal edge describes a deletion in A and each vertical edge describes an insertion of a character from B. There are additional diagonal edges at the positions, where the characters in A and B do not differ. The goal is now to get from the top left of the grid, to the bottom right with the least amount of vertices passed. This describes the previously mentioned shortest path problem.

Algorithm 1 shows the pseudocode provided by Myers [8] for finding the shortest path. It keeps track of the end points of multiple possible shortest paths in a vector V. Given the cost D that we are willing to spend, the algorithm checks for each of the paths how

4 https://git-scm.com/docs/git-diff

Algorithm 1 Pseudocode of the Myers diff algorithm [8]

```
    Constant MAX ∈ [0, M + N]
    Var V : Array[−MAX...MAX] of Integer
 1: V[1] ← 0
 2: for D ← 0 to MAX do
 3:     for k ← −D to D in steps of 2 do
 4:         if k = −D or k ≠ D and V[k − 1] < V[k + 1] then
 5:             x ← V[k + 1]
 6:         else
 7:             x ← V[k − 1] + 1
 8:         y ← x − k
 9:         while x < N and y < M and a_{x+1} = b_{y+1} do (x, y) ← (x + 1, y + 1)
10:         V[k] ← x
11:         if x ≥ N ∧ y ≥ M then
12:             Length of a shortest edit script is D
13:             Stop
14: Length of a shortest edit script is greater than MAX
```

far they get with horizontal edges (deletions) and vertical edges (insertions) costing 1 and diagonal edges (no change) not costing anything. D is initially set to 0 and increases in each interation (Line 2). In each iteration, the algorithm inspects the paths that end on diagonals represented by $k = x - y$. This means we can pass either one insertion (Line 5) or one deletion (Line 7) per iteration followed by several diagonal edges (Line 9). The algorithm ends when we find a path ending at the bottom right of the grid (Line 11). The first path to reach (N, M) is one with minimal cost since we iteratively increase D representing the cost. Therefore, the algorithm results in a shortest path.

Figure 2.2 displays an example of such a grid graph. The sequence "ABCAB" is transformed into "CBAB". Each blue edge represents a deletion or an insertion. The red edges represent identical characters in the two sequences. Each path from (0,0) to (5,4) equals a transformation from "ABCAB" to "CBAB". A shortest path is indicated in yellow and includes two deletions and one insertion.

After the shortest path is found, we are provided with a trace—a sequence of the vertices where there was no change from A to B—but not with an actual edit script. In the example in Figure 2.2, the trace would be (3,1) (4,3) (5,4). These tuples correspond to the subsequence "CAB" that both texts have in common. There is no need to compute the edit script since the trace already provides all the information necessary for our case.

The high-level idea behind the implementation is to compare the line in the blamed commit to the line in the parent commit. If the position we are currently investigating is in the trace, we know that there was no change at said position. Subsequently, if it is not in the trace it implies that there was an actual change at the given column. This information is then used to either confirm the returned `blame` information or to disregard it. Chapter 3 explains this in greater detail.

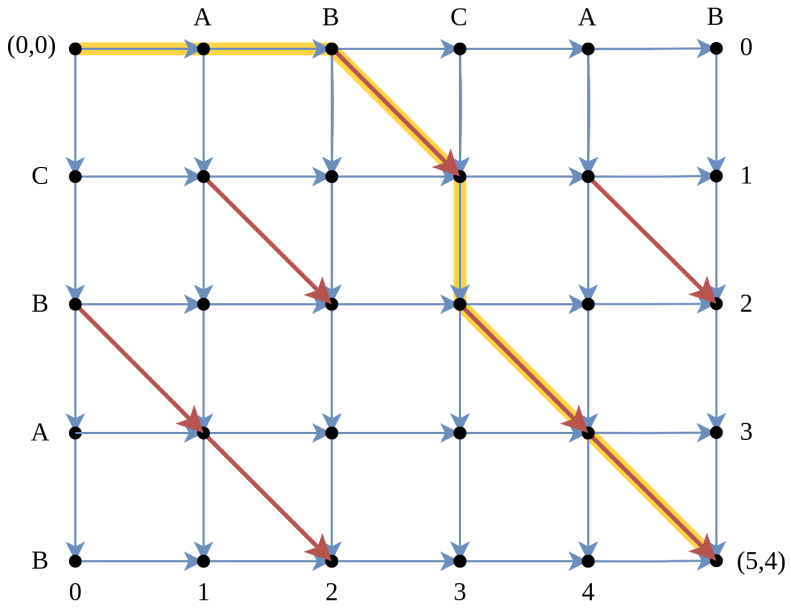

Figure 2.2: Myers grid for transforming "ABCAB" to "CBAB"

2.4 CLANG

The code that is written and easily readable by human software developers needs to be transformed into a form that a machine can understand and execute. This task is performed by compilers. LLVM is a commonly used compiler framework. A compiler built in LLVM consists of three parts: (1) a language-specific frontend that translates source code into LLVM-*IR*, (2) a language-independent optimizer for the *IR*, and (3) an architecture-specific backend that translates *IR* into machine code. Therefore, the problem of translating code written in a specific programming language into machine code is reduced to transforming it into LLVM-*IR*. CLANG is the frontend for C/C++, which means it translates C/C++ source code into LLVM-*IR*. LLVM then further helps with translating LLVM-*IR* into machine code.

Figure 2.3 shows a small C++ program and its corresponding *IR*. There are many different instructions which handle, amongst others, memory access, binary operations, and control flow. An overview can be found in the LLVM documentation[5].

There are several phases CLANG passes to generate LLVM-*IR* from source code. At first, the source code is transformed into a stream of tokens by the lexer. Secondly, this stream of tokens is taken and a concrete syntax tree (CST) is formed by the parser. Afterwards, there is a semantic analysis which creates an abstract syntax tree (AST) from the CST. Finally, CLANG translates the AST into LLVM-*IR* during the code generation phase. This translation is often referred to as lowering. In this last phase, VaRA attaches the additional metadata to the LLVM-*IR* instructions to have it available for further analysis. We use this to annotate the *IR* instructions with `blame` information.

One important aspect for this thesis is how source code locations are represented in AST nodes. That is because our `blame` algorithm requires the line in combination with the

5 https://llvm.org/docs/Reference.html

column number to look up the correct commit hash. We use functions that extract this information from the AST node's location to then pass the position on to our implementation of the character-based `blame` algorithm. This happens during the lowering of an instruction. The character-based `blame` algorithm, further explained in Chapter 3, uses this position in combination with the repository information to generate the commit hash of the commit introducing the specific character. The modified compiler then annotates the resulting commit hash as metadata to the LLVM-*IR* instruction.

```
1    int foo(int x) {
2        int result;
3        result = x + 42 - 1;
4        return result;
5    }
```

```
%1 = load i32, i32 %x.addr
%add = add nsw i32 %1, 42
%sub = sub nsw i32 %add, 1
store i32 %sub, i32* %result
```

Figure 2.3: Example translation from source code to LLVM-*IR*

2.5 VARA

With the help of the analysis framework VaRA one can efficiently build analyses on a high-level concept without having to deal with complicated low-level details. We can on a high-level specify to annotate instructions with commit hashes and VaRA provides the functionality to run a data-flow analysis on such annotated program files. For the annotations, Sattler et al. [10] introduce a modified Clang compiler[6] called Vlang. Vlang adds additional metadata to the *IR* instructions during the *IR*-code generation phase as explained in Section 2.4. An analysis can utilize this additional information while analyzing the resulting *IR* code. Figure 2.4 shows an overview of this process.

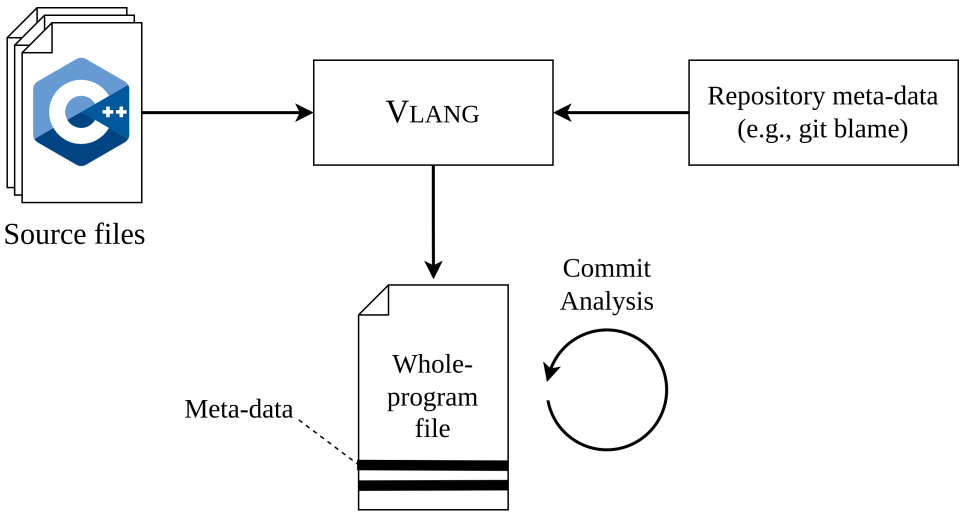

Figure 2.4: Simplified VaRA pipeline

6 https://github.com/se-sic/vara-llvm-project

One application of this framework is *commit-interaction analysis*. The goal of this analysis is to determine code from which commits interacts with each other on a data-flow level. For this analysis, VLANG annotates each instruction a commit hash that corresponds to the `blame` of the instruction. Afterwards, we use the annotated `blame` information as input to a modified data-flow analysis.

Sattler et al. [10] define commit-interaction graphs to represent the results of such a data-flow analysis. A commit interaction describes data-flow interactions between different commits (e. g., when code introduced by commit c_1 has a data-flow dependency on code introduced by commit c_2, we have a commit interaction between c_1 and c_2). The relation $\rightsquigarrow$ represents data-flow interactions between instructions. The function *base* maps an instruction to the commit that last modified it, i. e., its `blame` annotation. Commit interactions are defined as follows:

$$CI(i_1, i_2) = (base(i_1), base(i_2)) \;\; with \;\; base(i_1) \neq base(i_2) \;\; \wedge \;\; i_1 \rightsquigarrow i_2$$

On the *IR* level, a more fine-grained mapping of commit hashes is possible compared to the currently implemented line-based approach. With the line-based `blame` we can only determine, which commit last modified a full line. If only a small section of a line is modified by a commit, all instructions corresponding to the line get assigned the new commit. In Chapter 3, we introduce a `blame` algorithm that works on a character basis. We use this information to then annotate the *IR* instructions with AST-based `blame`. The annotated `blame` information is essential for the data-flow analysis to uncover commit interactions.

A commit-interaction graph further aggregates and summarizes commit interactions within a software project. It is a directed graph with the interacting commits ($\mathcal{C}$) as nodes and all commit interactions of a program ($\mathcal{CI}$) as edges, formally written as CIG $= (\mathcal{C}, \mathcal{CI})$. With the help of such graphs, we can answer questions related to the development of software projects, e. g., what commits affect central code. "Central" in this case means central in the data-flow dependency structure of a program. We can easily extract this information from a CIG since it represents the impact commits have on the data-flow dependency structure of a program. The node degree of a commit within a CIG describes how many other commits it interacts with and, therefore, is a good indicator of whether the code introduced by the commit holds a central role within the program. Making the `blame` annotation more precise influences these results since the data-flow analysis relies on the `blame` information. Section 5.3 further investigates these effects.

3 IMPLEMENTATION

In this chapter, we discuss AST-based `blame` annotations. At first, we describe how the VLANG compiler annotates the *IR* instructions and what information we can use for the `blame` computation. Afterwards, we introduce a character-based `blame` algorithm with which VLANG is, subsequently, able to annotate the instructions on an AST basis. Therefore, we refer to the annotations as AST-based `blame` and they are computed using a character-based `blame` algorithm.

3.1 COMPILER INTEGRATION

To create the AST-based `blame` annotations, we need to assign commit hashes to LLVM instructions. We achieve this via a modified CLANG compiler. As mentioned in Section 2.5, VLANG can add additional metadata to the *IR* instructions. We use this ability to add `blame` annotations to the *IR*. This annotation happens during the code generation phase.

In this phase, the compiler translates an AST into LLVM-*IR*. The AST nodes contain information about their source code locations. These locations point to the tokens in the source code that correspond to the AST nodes. When generating the *IR* instructions for an AST node, we insert calls to `git blame` to annotate the instructions with `blame` information. With the help of the location from the AST node, we can compute the commit hash that introduces the token(s) that correspond to the instructions.

Previously, this computation and annotation only used the line number. To make the `blame` information more precise, we use line and column information to character-wise compute the commit hashes (Section 3.2). To sum it up, we compute the `blame` for each relevant AST node to then annotate the corresponding instructions with `blame` information during the *IR*-code generation. Since the CLANG AST uses a hierarchy for its different node types, we can ensure that each node type and the corresponding code generation have at least one call to the `blame` computation.

3.2 CHARACTER-BASED BLAME ALGORITHM

We implement our AST-based commit mapping feature as part of the VLANG compiler. We introduce the flag `-fvara-ast-GB` to the VLANG compiler which enables the AST-based blame annotations. During the code generation phase, a function adds the commits calculated by the character-based `blame` algorithm as metadata to the LLVM-*IR*. These annotations are no longer based on a single character and we therefore refer to them as AST-based `blame` annotations. To implement this algorithm, we need a function that given a repository, file path, line number, and column number, computes the commit which introduced the character at the given location. This differs from the current implementation by using not only the line information but also the column information. Algorithm 2 outlines how we

compute the character-based `blame`. The main idea is to follow the history of the character which is being investigated back until the point where it was first introduced.

Algorithm 2 Character-based commit mapping

```
1: CurrentBlame := Line-based blame
2: (CurrentLine, ParentHunk) := Extract the relevant lines from CurrentBlame and from its
   parent commit
3: Compare CurrentLine to ParentHunk
4: if there has been a change at the given location then
5:     return CurrentBlame
6: else
7:     CurrentBlame := Line-based blame from parent commit (HEAD)
8:     Go to 2
```

We use the commit hash that is returned by the line-based `blame` as a base commit (Line 1). Then, we extract the lines corresponding to the investigated character in *CurrentBlame* (:= *CurrentLine*) and its parent commit (:= *ParentHunk*) (Line 2). *CurrentBlame* has either modified or introduced this line. Next, we compare *CurrentLine* to *ParentHunk* to determine whether the individual character was actually changed by the commit *CurrentBlame* (Line 3). For the comparison, we use an adjusted version of the Myers algorithm further explained in Section 3.2.2. If we ascertain an actual change, the *CurrentBlame* commit introduced the investigated character, so we return *CurrentBlame* (Line 5). Otherwise, we know that the character was not changed by *CurrentBlame*, so we go one step further back in the commit history. We again set *CurrentBlame* to be a line-based `blame` but now only considering the history from the parent commit on backwards (Line 7). HEAD is a reference to the commit that is considered the current version of the project. In Line 7, we set it to refer to the parent commit. When we then compute the line-based `blame` we only consider the repository history from the HEAD commit on backwards. Initially, HEAD usually points to the most recent commit. This procedure is done iteratively until we find the commit that actually introduced the investigated character.

3.2.1 *Keeping Track of Correct Location*

We have different situations in which the location of the investigated character changes. It is not guaranteed that the line number from our initial state is the same as the line number in the *CurrentBlame* commit. There might have been insertions or deletions before the given position. The same problem occurs when we investigate the parent of *CurrentBlame*. And in the case that we determine that we have no actual change from the *CurrentBlame* to its parent commit (Line 6), the column information potentially is also different in the parent commit. At this point, we know that the line contains some modification, so there might have been insertions or deletions before the investigated character. Therefore, it is very important and necessary to keep track of the location of our investigated character.

Before we can extract the relevant lines that are used in the comparison (Line 2), we need to compute the correct line number in *CurrentBlame*. The content of the line cannot have been changed between the HEAD commit and the *CurrentBlame* commit because we are

looking at the line-based blame. The line-based `blame`, by definition, is the commit that introduced the content of the line. So we simply find the line in *CurrentBlame* and return the line number. Hence, we can easily extract the line of *CurrentBlame*.

This is more difficult for the parent commit. We do not know whether a similar line that was only minorly changed is present in the parent commit or not because *CurrentBlame* could have fully introduced and not only modified the line. Therefore, we look at the diff hunks from the current commit to its parent. We determine in which diff hunk the relevant text sequence is most likely located and continue working with that diff hunk (*ParentHunk*). Now that we have a range of lines possibly containing the investigated line, we still need to decide whether *CurrentBlame* newly introduced the line. If that is the case, no similar line is contained in the *ParentHunk*. Additionally, if that is not the case, we need to compute the exact location in the parent commit. We decide and compute these aspects when comparing *CurrentLine* to *ParentHunk* with the help of an adjusted Myers algorithm explained in Section 3.2.2.

3.2.2 *Adjusted Myers Algorithm*

In Line 3 of the character-based `blame` algorithm, we compare multiple lines $a \in A$ with a single line b. Hence, we can not simply use the Myers algorithm as explained in Section 2.3 because it only compares two text sequences. Besides determining whether there is a change at the given position, we additionally need to identify which line from the multiple lines A was possibly modified into the single line b and what the column information of the character in this previous line is. This is not included in the traditional Myers algorithm.

To determine which line in the parent hunk is the interesting one, we run the Myers algorithm on each pair of lines (a, b) with $a \in A$. The Myers algorithm computes the minimal number of insertions and deletions needed to transform line a into line b. During the algorithm, we additionally keep track of the traces of each possible path and finally return the one corresponding to the shortest edit script. For all the characters that do not change in the transformation from a to b, the trace contains a pair with the column information of the character in a and in b. We look for the line $a' \in A$ that results in the longest trace T, since it represents having the most characters in common with b. This gives us the line number we might use in a following iteration.

The trace T contains all pairs $(c_{a'}, c_b)$, where $c_{a'} \in \mathbb{N}$ refers to a column number in a' and $c_b \in \mathbb{N}$ to a column number in b, for which the character at the column numbers are equal in a' and b. Let $y \in \mathbb{N}$ be our investigated position, then we know that there was no actual change at y if $\exists x \in \mathbb{N} : (x, y) \in T$. In that case, we know by the definition of the trace that the investigated character was not modified. Finally, if it is included in the trace, we simply look up the corresponding column position for a' which is x. This is the column number we use in the following iteration.

Figure 3.1 illustrates this procedure for the example in Figure 1.1. In this example, we do not have multiple lines in *ParentHunk*, so we simply compare the line b from *CurrentBlame* to line a' in its parent commit. T is the trace we obtain when running the Myers algorithm on the two given lines. For each character that is not changed when translating a' to b, it contains a pair of column numbers. If we investigate the character '1' which has the column information of 19, we see that $\nexists x \in \mathbb{N} : (x, 19) \in T$. This shows us that there has been a

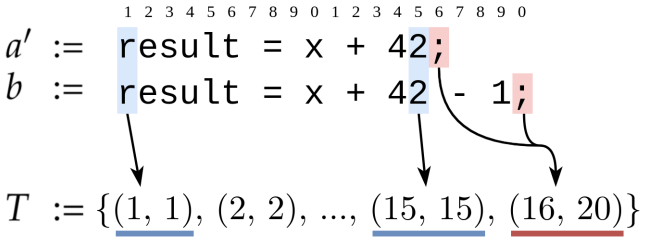

Figure 3.1: Example for the adjusted Myers algorithm

change at the position. If we take a look at the semicolon at column number 20, we find an entry that fulfills the condition $\exists x \in \mathbb{N} : (x, 20) \in T$ with $x = 16$. This tells us that the character itself did not change from a' to b but the location of the character changed. Therefore, we need to continue investigating with column number 16 instead of 20.

3.3 LIMITATIONS

While this character-based `blame` algorithm can handle many common change-scenarios, our implementation has some limitations that we address in the following. One of the primary constraints of the implementation lies in the character basis on which we operate. We only examine the characters present in the token of the most recent version of the software. Hence, we can only determine their origin and not inspect characters that possibly were removed from a token. For example, this occurs when a variable name is shortened. This commit will be overlooked by the character-based algorithm, since the algorithm determines that all the characters that we examine were previously present. To solve this problem, we would need to further information about the token to the character-based `blame` algorithm. This would require us to make many adjustments to the algorithm that deal with determining whether the token changed whenever we iterate. In this thesis, we focus on a character basis. This possible improvement of the algorithm is out of the scope of this thesis and is left for future work.

The AST-based `blame` annotations highly depend on the location which we provide as a parameter to the character-based `blame` algorithm. As discussed in Section 3.1, we might use imprecise or too wide-ranging location information in our `blame` computation, since we only hook into the code generation phase of Clang. On the one hand, if we include more characters from the source code than relevant for the investigated instruction, we might obtain a commit that is more recent than the commit that actually introduced it. On the other hand, if we include less characters, we possibly return a commit that is older than what we would expect. However, we can be sure that the location information is not completely off since we cover all base node types in the AST hierarchy of Clang.

Some steps of our algorithm rely on Git's default diff information and the Myers algorithm. Therefore, flaws in the extraction of the diff hunk and in the Myers algorithm will lead to errors in the character-based calculation of the `blame`.

Finally, an important aspect to bear in mind is the limitation associated with the performance. With the line-based approach, the annotation is already quite expensive because we have a `blame` lookup for every relevant AST node. Caching the `blame` would certainly

improve the overall performance. Now for the AST-based approach we iterate over each character in a token and for each we have at least one but possibly multiple `blame` lookups because of the iterative algorithm. Additionally, we need to consider that in each step of the iteration we use the Myers algorithm to determine whether we have an actual change at the inspected position. All these aspects lead to a significant runtime overhead.

4 METHODOLOGY

This chapter describes the methodology through which we evaluate the AST-based `blame` functionality. We introduce our research questions in Section 4.1. Afterwards, Section 4.2 presents an overview of the test repositories and programming constructs we use to evaluate the AST-based `blame` information. In Section 4.3, we make some remarks about the baseline used to evaluate RQ1.1.

4.1 RESEARCH QUESTIONS

The introduction of more fine-grained `blame` information should help with answering a variety of questions. Having precise information about the source of certain code segments is beneficial in the context of authorship attribution as well as in commit-interaction analyses. We need to ensure that the AST-based `blame` information provides an improvement in precision compared to the the default line-based `blame` information. Therefore, we take a look at how well the introduced implementation works as well as what the effects are on a existing commit-interaction analysis. We want to answer the following questions.

RQ1 **How much do AST-based commit mappings improve `blame` information on an *IR* level?**

The overall goal of the thesis is to figure out how we can improve the `blame` information on the LLVM-*IR* level. We want to get insights about how the introduced algorithm changes the precision of the `blame` information. Therefore, we split RQ1 into the following two sub-questions, approaching the problem qualitatively and quantitatively.

RQ1.1 **Which common code development scenarios benefit from AST-based commit mapping?**

RQ1.2 **How many instructions are mapped differently in real-world projects with AST-based `blame` annotations compared to the line-based `blame`?**

The most important aspect we need to investigate is the correctness of the AST-based commit annotations. Hence, for RQ1.1, we examine a few smaller projects for which we manually create a baseline. The baseline consists of a mapping of each instruction to the commit hash that we expect from a human standpoint. We compare the results from the implementation to the manually derived baseline. These smaller projects represent common changes occuring during code development. By working on such small projects, we have the possibility to review each *IR* instruction individually. It helps us to see which scenarios benefit from the more fine-grained `blame` information.

Additionally, we need to decide whether these AST-based commit annotations significantly differ from the line-based annotations returned by `git blame`. For that reason, to answer RQ1.2, we use real-world projects, which helps us gain an insight about the impact

of AST-based `blame`. We investigate the total difference between line-based and AST-based annotations on said real-world projects.

There are different program analysis techniques as well as repository mining techniques to gain a better insight into program development. To estimate the impact AST-based `blame` can have on modern analyses, we ported a recently published commit-interaction analysis to AST-based `blame`, asking the question:

RQ2 **What is the impact of AST-based `blame` information on commit-interaction analysis?**

To evaluate RQ2, we take a closer look at the impact of the more precise AST-based commit annotations on the results of a commit-interaction analysis proposed by Sattler et al. [10]. It uses commit annotations, such as the ones discussed in this thesis, to detect commit interactions and combines it with the data-flow dependency structure of the program. This helps with better understanding code development. One interesting aspect is the identification of the commits that affect central code. Central code is "code which interacts with lots of other code" [10]. The analysis uses the `blame` annotations made by VLANG to identify commits that have a central position in the dependency structure of the analyzed program. The changes of the `blame` annotations can therefore lead to different or new commits being identified as central to the software project.

4.2 TEST REPOSITORIES

This section briefly describes the repositories used to evaluate the research questions. For the small test repositories used for RQ1.1, we examine the functionality each repository provides, the programming constructs being used, and the changes made to the code. Table 4.1 sums up the programming constructs that we further investigate in each repository. C++ is a complex language with many different constructs and many different AST nodes. Because of the large amount of AST nodes, we can not provide a set of test cases that includes all nodes. For that reason, we focus on very common programming constructs. In the following descriptions, a separate consecutive commit introduces each mentioned alteration.

4.2.1 *Case 1: Arithmetic Expression*

At first, we inspect a simple change in an arithmetic expression (see Figure 4.1). This is almost identical to the example which is used in Chapter 2. We have an expression in which a variable is assigned a value. That value is computed with multiple binary operators. For this test repository we take a closer look at these binary operators.

```
result = x + 42;          →   5   result = x + 42 - 1;
```

Figure 4.1: Summary of *Case 1: Arithmetic Expression*

The history of the repository (A.1.1) is rather short. We added a binary operator to the computation of *result*. To be precise we added the `- 1` to the expression. Since the different operators can be nicely differentiated in the IR we want to assign the appropriate hashes to

Id	Name	Programming elements	Repository history
1	Arithmetic Expression	Binary operators (arithmetic)	A.1.1
2	Conditional	Binary operators (comparison)	A.1.2
		If-statements	
		Conditional declarations	
3	Class	Class constructor	A.1.3
		Member functions	
		Object creation	
		Dynamic memory allocation (`new`)	
		Dereference (pointer)	
4	Function Call	Function calls	A.1.4
5	Function Declaration	Function signatures	A.1.5
		Function bodies	
6	Loop	Basic `for`-loop	A.1.6
		Range-based `for`-loop	
		Increment operator	
7	Namespace	Using namespace	A.1.7

Table 4.1: Overview of test repositories

each instruction corresponding to these operators. Therefore, in this case we investigate the addition of a binary operator.

This change in source code introduces one addtitional instruction in the IR. In Figure 4.2 we see all instructions belonging to the one line of code. The change only introduces one additional instruction, namely the one with the subtraction operation. In this case it is clear that we only want to assign the newest commit to the new instruction. For the AST-based `blame`, we expect the remaining instructions to be assigned to the previous commit.

```
  %1 = load i32, i32* %x.addr
  %add = add nsw i32 %1, 42
+ %sub = sub nsw i32 %add, 1
  store i32 %sub, i32* %result
```

Figure 4.2: Change of the IR in *Case 1: Arithmetic Expression*

4.2.2 *Case 2: Conditional*

In the next repository, we look at two different `if`-statements. The first is a conditional declaration (see Figure 4.3). Here, we investigate an alteration in the variable name, followed

by a modification of the type of the newly introduced local variable. Two consecutive commits introduce these changes in the aforementioned order.

```
if (int y = x * 2) {          →   4   if (float result = x * 2) {
      return y;               →   5       return result;
                                  6   }
```

Figure 4.3: Summary of *Case 2: Conditional* - Conditional declaration

The other programming construct investigated in this repository is a simple `if`-statement (see Figure 4.4). In this case, we modify the comparison operator used in the condition. In the previous test case, we also look at a binary operator but only the addition of one and not a change in an already present one.

```
if (x < y) {                  →   11  if (x <= y) {
                                  12      return true;
                                  13  }
```

Figure 4.4: Summary of *Case 2: Conditional* - Simple `if`-statement

4.2.3 *Case 3: Class*

This test repository represents the class construct in C++ (see Figure 4.5). In the declaration of the class `Foo`, we change the name of a member variable, the name of a paramater of the constructor, and the constructor itself. This is again done consecutively in the aforementioned order.

```
                                  3   class Foo {
int i;                        →   4       int value;
                                  5   public:
Foo(int i) : i(i) {};         →   6       Foo(int param) : value(param) {
                                  7           setVal();
                                  8       };
                                  9
int getVal() { return i; };   →   10      int getVal() { return value; };
                                  11      void setVal() { value = 42; };
                                  12  };
```

Figure 4.5: Summary of *Case 3: Class* - Class declaration

We also consider different approaches to object creation (see Figure 4.6). For this, we introduce many modifications to the source which we omit for brevity. The full history of the repositories can be found in the Appendix A. By introducing many small changes, we are able to thoroughly test the character-based `blame` that is introduced in Chapter 3. Interesting aspects in this case are the introduction of the allocation with `new`, correctly tracing the origin of variable use and also the unary operator for dereferencing (`*`).

```
Foo fooA = Foo(21);          →  15  Foo* fooA = new Foo(21);
                                16  Foo fooA2 = *fooA;
Foo fooB = Foo(42);          →  17  Foo* fooB = new Foo(42);
fooA.setVal();               →  18  fooB->getVal();
```

Figure 4.6: Summary of *Case 3: Class* - Object creation

4.2.4 *Case 4: Function Call*

Next, we investigate another very small test case. There is an alteration in the call parameters of a function call (see Figure 4.7). The function takes two parameters in total and we only change the value of one. Important to note, the call parameters are constants and not variable. This plays a big role in the translation to the *IR*, as for variable call parameters each one would get separate *IR* instructions. In the case of constant call parameters, this does not happen and the translation of the function call only consists of one *IR* instruction.

```
some_function(4, true);      →  8  some_function(4, false);
```

Figure 4.7: Summary of *Case 4: Function Call*

4.2.5 *Case 5: Function Declaration*

This scenario highlights changes in function signatures (see Figure 4.8). We look at a modification in a parameter name followed by an addition of a new parameter. Therefore, we have three different commits, including the base commit initially introducing the function, interacting with a single line. With line-based `blame`, we can only assign one commit because we examine single line. This commit has to be the most recent one, because it is the one last modifying the line. Therefore, we lose the information about the impact of the two other commits.

```
int some_function(int x) {   →  3  int some_function(int result, char y) {
    return x;                →  4      return result;
                                5  }
```

Figure 4.8: Summary of *Case 5: Function Declaration*

4.2.6 *Case 6: Loop*

Next, we look at `for`-loops. In a range-based `for`-loop (see Figure 4.9), we first change the type from `auto` to `int`. Then, we rename the array used in the loop. Finally, we also rename the loop variable. These modifications provide us again with multiple commits interacting with a single line. Once again, with line-based `blame` we can only assign a single commit to

the instructions corresponding to that line. We expect our AST-based implementation to be able to include each commit influencing the line in the annotations.

```
int array[] = {1, 2, 3, 4};          →   4  int numbers[] = {1, 2, 3, 4};
                                         5  int result = 0;
for (auto n : array) {               →   6  for (int element : numbers) {
    result += n;                     →   7    result += element;
                                         8  }
```

Figure 4.9: Summary of *Case 6: Loop* - Range-based `for`-loop

We also consider a simple `for`-loop in which we also rename the loop variable (see Figure 4.10). Another interesting programming element we encounter here is the increment operator (++). CLANG handles each component in a dedicated manner. Consequently, this operator is also processed differently than binary operators.

```
for (int i = 0; i < 4; i++) {    →  14  for (int count = 0; count < 4; count++) {
    result += i;                 →  15      result += count;
                                    16  }
```

Figure 4.10: Summary of *Case 6: Loop* - Basic `for`-loop

4.2.7 *Case 7: Namespace*

This test scenario uses the namespace construct (see Figure 4.11). We call a function which takes an element of a namespace as a parameter. After introducing the namespace with the `using` keyword, we no longer need to use the namespace scope resolution operator (`::`). This is only a syntactical change, so the AST should not be influenced by this change. Subsequently, it is difficult to say what to expect in this case. If we say, we want to only consider the changes to the AST, we expect the result to be the previous commit hash. Otherwise, if we base our decision on the token we expect the result to equal the line-based `blame`.

```
                                     7  using namespace my_namespace;
                                     8
                                     9  int main() {
printf("%d", my_namespace::x);   →  10      printf("%d", x);
                                    11  }
```

Figure 4.11: Summary of *Case 7: Namespace*

4.2.8 *Real-World Projects*

To evaluate RQ1.2 and RQ2, we use the real-world projects used by "SEAL: Integrating Program Analysis and Repository Mining" [10]. These projects are 13 C/C++ projects

chosen from different application domains: BISON, BROTLI, CURL, GREP, GZIP, HTOP, LIBPNG, LIBSSH, LIBTIFF, LRZIP, LZ4, OPUS, and XZ. The mentioned paper provides results of a central code analysis with line-based `blame` information for these 13 projects. Subsequently, we can easily compare these results to the ones we obtain using AST-based `blame` information, to determine the impact our fine-grained AST-based algorithm has on the calculation of central code.

4.3 BASELINE

For the small test repositories that cover basic programming constructs, we manually create a baseline. This baseline describes the expected results from our AST-based `blame`. While creating this baseline, we have to consider some limitations of the AST-based `blame` implementation. We map an instruction to the commit introducing the functionality represented by the instruction but also keeping the source code token relating to the instruction in mind.

For example, a variable occurring in a program is associated with a set of instructions. If we change the variable name, many of the instructions connected to that variable change and some remain the same. Ideally, we would assign the different instructions to the new commit and the unchanged ones to the old commit. However, since the token we inspect for the instructions is just the one token of the variable name, we can not differentiate the instructions in such a case.

Now, we change the variable type, e.g., from an integer to a float. The rest of the program remains the same. In the IR, we see this type change in any variable occurrence. Since the token in the source code did not change, we do not assign a new commit hash to each instruction relating to the variable. We only assign the new commit hash to the instructions regarding the declaration of the variable.

5 EVALUATION

In this chapter, we present the results of our evaluation of the AST-based `blame` implementation and we answer the research questions presented in Section 4.1. First, we investigate what benefits AST-based `blame` annotations can have on different code development scenarios. Afterwards, we look at the differences between the line-based and the AST-based `blame` annotations in real-world projects. Finally, we examine what impact of the more fine granularity have on the results of a specific commit-interaction analysis, namely a central-code analysis.

5.1 BENEFITS ON COMMON CODE DEVELOPMENT SCENARIOS

In this section we evaluate RQ1.1 and investigate which code development scenarios can benefit from an AST-based `blame` approach. Section 4.2 presents an overview over the test repositories used for the evaluation.

5.1.1 *Evaluation of Test Repositories*

Figure 5.1 shows an overview of how many instructions where correctly annotated by the line-based `blame` (in red, the left bar for each case) versus the AST-based `blame` (in blue, the right bar for each case). The baseline is shown behind the the red and blue bars in grey, and represents the total number of instructions in the respective test case that get assigned `blame` information.

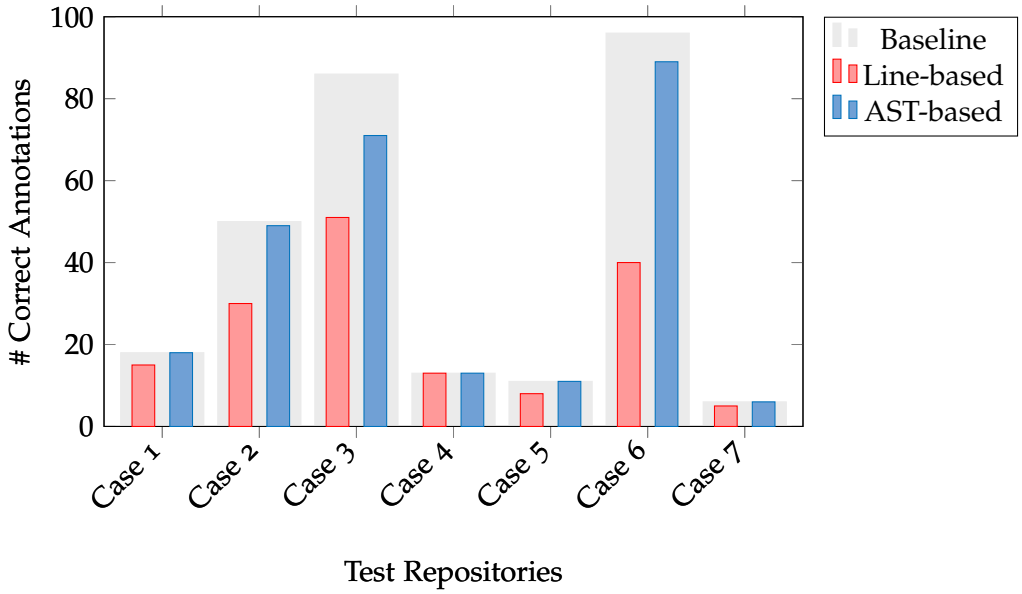

Figure 5.1: Correct `blame` annotations for line-based and AST-based approaches.

We notice the biggest improvement in correct annotations in *Case 6*. This is also the test case with the most instructions in total. The only test case where we see no improvement is *Case 4*. The test case with the least instructions in total is *Case 7*. We group the seven test repositories into three different categories.

1. The AST-based `blame` shows an improvement to the line-based `blame` and is now equal to the baseline. **(Cases 1, 5, and 7)**
2. The AST-based `blame` shows an improvement to the line-based `blame` but is not equal to the baseline. **(Cases 2, 3, and 6)**
3. Both the AST-based and the line-based `blame` are equal to the baseline, so there is no improvement possible. **(Case 4)**

We now further inspect each category.

Category 1

This category is about the cases where the AST-based `blame` shows a significant improvement to the line-based one and is now equal to our manually derived baseline. The first test case falling into this category is the one regarding arithmetic expressions (*Case 1*). The interesting change in this case was the addition of a binary operator. With line-based `blame`, we were unable to differentiate the other operators that where previously in the same line from the newly added one. This is now solved and only the added operation is assigned to the new commit.

The second test case in this category is the one regarding function declarations (*Case 5*). With the more fine-grained `blame` information, we now can differentiate between the different parameters in a function signature if they are in the same line. The introduced AST-based implementation is able to individually assign `blame` information to each parameter.

The last test case in this category is the one dealing with namespaces (*Case 7*). This is a special case because the introduced change is only syntactical and thus not visible in the LLVM-*IR*. We have a function call where one parameter is a variable from a namespace. There are two *IR* instructions that are interesting in this case: the loading of the variable `x` from the namespace `my_namespace` and the `printf` function call (Figure 4.11, Line 10). The improvement that we gain in this case is that we now correctly assign the `printf` function call with the commit before the introduction of the `using namespace` construct. For the instruction corresponding to the variable use from the namespace the expected result is not as clear. On the one hand, it is acceptable to assign the commit that introduced the `using namespace` construct because that way we do not lose the information of the change in the *IR*. This is what is done by the line-based `blame`. On the other hand, it is valid to assign the commit before the change, since the *IR* does not change. This is what is done by the AST-based `blame`.

Category 2

In this category, we find the test cases where we see an improvement to the line-based `blame` but there is still room for improvement to the baseline. First, let us examine the improvements we see in these cases.

With the AST-based implementation in the test repository dealing with conditionals (*Case 2*), we are able to correctly differentiate a change in a conditional declaration in the form of a type change and a variable name change. If a commit only changes a binary operator, like the comparison operator in the simple if-statement, we also only assign the instruction dealing with that operator to the new commit.

In the class test repository (*Case 3*), we see a significant improvement in correct annotations, but we also have the highest difference from the baseline. Changes in class attributes, member functions, and member function calls benefit from the more fine-grained algorithm. We attribute this to the fact that we inspect individual tokens and not whole lines. It is often the case that different *IR* instructions correspond to a single line. An example of this is a class constructor with an initializer list. If we investigate the source commit of the individual tokens instead of the whole line, the results are more specific to the corresponding instruction.

In the test repository about loops (*Case 6*), we see the biggest improvement in correct `blame` annotations. It is also the repository with the most instructions in total. The reason for this are the many instructions needed for initialization/cleanup of the loop control variables, condition checks, increment/decrement operations and loop body execution. Since initialization, condition and increment/decrement operations in a for-loop are often found in the same line, we can greatly improve the accuracy of the `blame` annotations by switching from a line-basis to the AST-basis.

This category is characterised by the fact that not all AST-based commit annotations are equal to the baseline. In the conditional repository, it is only one instruction. It is an allocation of a `%cleanup.dest.slot` in the context of the conditional declaration. This stores a cleanup flag that indicates whether certain cleanup actions need to be performed. There is no specific token in the source code which corresponds to this instruction.

The aspect not correctly working in the class test repository is the assignment of instructions of landing pads. After the introduction of a memory allocation with `new`, a corresponding set of instructions named `lpad` is introduced. This landing pad contains instructions dealing with exception handling[1]. It would be expected that all instructions in these landing pads are assigned the same commit. However, both the line-based and the AST-based `blame` assign parts of the instruction set to a different commit hash. A similar problem arises for other instructions related to exception handling. We again have no concrete token in the source code which represents these instructions.

In the loop example, we face the same problem. The loop variables `%__range` and `%__end` and their occurrences throughout the LLVM-*IR* are assigned to wrong commit hashes. In the source code, the two loop variables do not have a corresponding token.

Because the AST-based algorithm works by computing the `blame` of characters, we obtain these observed faulty annotations. We have no characters that represent the instructions. Therefore, we can not correctly compute the `blame` annotations for these instructions.

Category 3

Now, we look at the test case where the line-based results and the AST-based results both equal the baseline. It is the one regarding function calls (*Case 4*). We investigate a change

1 https://llvm.org/docs/ExceptionHandling.html

in the call parameters in this repository. The call parameters consist of constants. Due to this fact, VLANG translates the function call to a single LLVM instruction. To not loose information about the change in the call parameters, we need to assign the newer commit to that instruction. Subsequently, we do not improve from the line basis, since there is no room for improvement.

5.1.2 *Discussion*

In almost all cases, we observe an improvement compared to the line-based commit annotations. We see the highest difference in the test repository regarding loops (*Case 6*). The only repository in which we do not have any room for improvement is the one concerning a change in constant function call parameters (*Case 4*).

We attribute the reasons for wrong AST-based annotations to one factor. LLVM-*IR* instructions do not necessarily have a corresponding token in the source code. This is a limitation when it comes to the character-based `blame` computation. We have no locations to compute the correct commit hashes for the annotations. However, we can say that all wrong annotations in our test cases are equal to the annotations from the line-based `blame` or assign a commit lying between the line-based and the correct value. There is no annotation of a commit older than the line-based one. Therefore, the AST-based annotations are not worse than the line-based ones.

To address RQ1.1, we can conclude that almost all the code examples from our test cases benefit from AST-based `blame` annotations. The only test case not benefitting from the added fine granularity is *Case 4*.

5.2 STATISTICS OF DIFFERENCE TO LINE-BASED BLAME

In this section, we answer RQ1.2. That is, we investigate the quantitative difference between the line-based and the AST-based `blame` annotations. We compile real-world projects once with line-based `blame` annotations, once with AST-based `blame` annotations, and then look on the total difference between the annotations. Additionally, we compare the AST-based `blame` annotations to line-based `blame` computations based on the line provided by debug information.

5.2.1 *Results*

We were unable to compute the AST-based `blame` for all of the 13 real-world projects. The reason for this is that the computations were too time consuming. Within a timeframe of 9 days we were only able to generate results for the projects BROTLI, HTOP, LRZIP, and XZ. Therefore, we have no results for the projects BISON, CURL, GREP, GZIP, LIBPNG, LIBSSH, LIBTIFF, LZ4, and OPUS.

Figure 5.2 shows the results from comparing the AST-based `blame` annotations against **(a)** the line-based `blame` annotations and **(b)** line-based `blame` computations based on debug locations. We further refer to **(a)** as *AntDiff* and **(b)** as *DbgDiff*. For each bar chart, 100% represents the total number of instructions with `blame` annotations. Note, as we

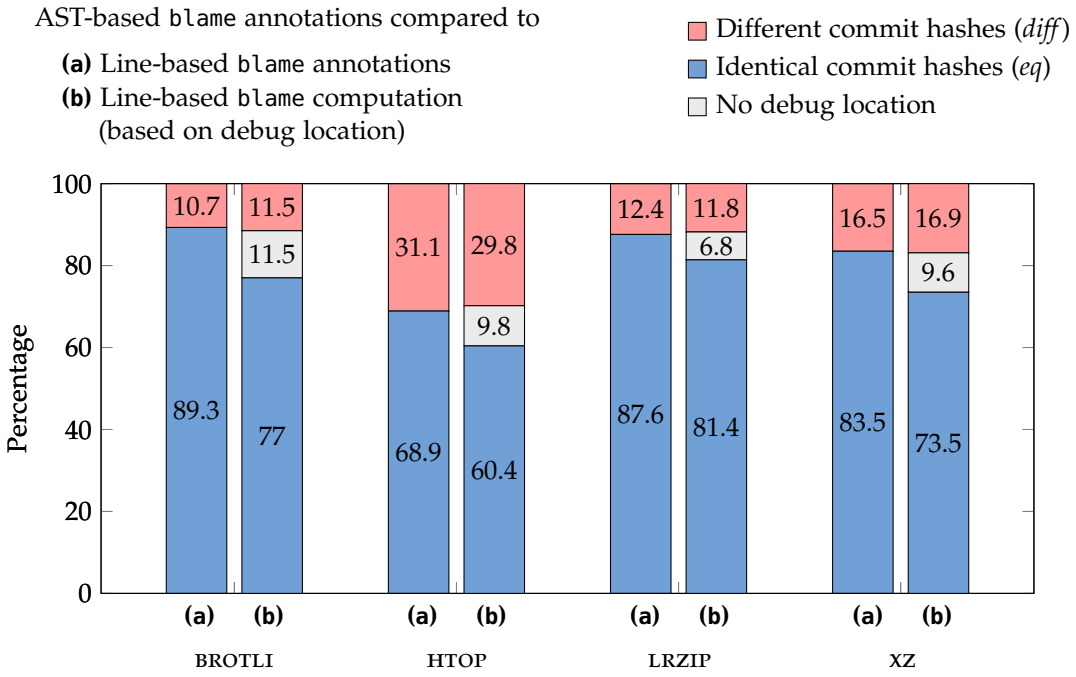

Figure 5.2: Overview of quantitative difference between AST-based and line-based `blame`

compare against line-based `blame` computations based on a debug location in *DbgDiff*, not all instructions that are annotated with `blame` information have a debug location. Hence, we can only compare the subset of the instructions with debug location, seeing overall projects this affects 6.8% to 11.5% of the instructions with `blame` annotation.

The percentage of instructions that are equally assigned (*eq*) range between 60.4% and 89.3% (an average of 77.7%). Subsequently, we observe differently assigned instructions (*diff*) in 10.7% to 31.1% of the instructions (an average of 17.6%). The project HTOP shows the highest difference between AST-based and line-based `blame`. Table 5.1 shows that it is also the project with the longest runtime. Another interesting observation is that BROTLI, the project with the highest total number of instructions, is the project with the lowest difference between AST-based and line-based `blame`.

The difference between *AntDiff* and *DbgDiff* for *diff* ranges from 0.4% to 1.3% (average of 0.8%). For BROTLI and XZ, *diff* is slightly higher for *DbgDiff*, whereas for HTOP and LRZIP, the percentage is slightly lower. The percentage of *eq* is smaller for *DbgDiff* in all cases.

	diff		*eq*		AST-based
	(a)	**(b)**	**(a)**	**(b)**	runtime
BROTLI	8,845	9,475	73,734	63,617	72.1 h
HTOP	15,009	14,345	33,209	29,113	104.8 h
LRZIP	9,296	8,847	65,681	61,057	63.4 h
XZ	1,591	1,628	8,063	7,100	69.5 h

Table 5.1: Difference between AST-based and line-based `blame` in numbers of instructions

5.2.2 *Discussion*

In Section 5.1.2, we establish that many different code scenarios benefit from the AST-based `blame`. This explains why we notice a significant difference between the AST-based and the line-based results. Without further information about the implementation of the individual projects, we can not explain the difference in the percentages between the projects. However, for each instruction not assigned to the line-based `blame`, we have to look further back in the history of the repository. Due to this iterative nature of the character-based `blame` algorithm, we significantly prolong the runtime. Consequently, it is logical that we observe the highest runtime in the project with the highest difference (HTOP).

We observe a slightly higher percentage of *diff* for BROTLI and XZ *DbgDiff* compared to *AntDiff* because the line-based `blame` annotations do not 100% correspond to the computation based on the debug location. The slightly lower percentage we see in the projects HTOP and LRZIP is explained by the presence of instructions without debug location. This is also the reason why the percentage of *eq* is smaller for *DbgDiff* in all cases.

All projects from which we obtained results took over 2.5 days to compile with AST-based annotations. Nine out of 13 projects took longer than nine days to compile. We already mentioned in Section 3.3 that the added fine granularity leads to an overhead in the performance. Subsequently, we observe this runtime overhead.

To answer RQ1.2, we can say that on average 17.6% of the instructions in the analyzed real-world projects are mapped differently with AST-based `blame` annotations compared to the line-based `blame`. This is quite the significant percentage. This observation combined with what we obtain in Section 5.1 indicates that AST-based commit mappings could have a noticeable, positive influence on `blame` information on an *IR* level. Raising further questiosn on whether this indication proves to be true.

5.3 IMPACT ON COMMIT-INTERACTION ANALYSIS

To answer RQ2, we perform a central-code analysis on program files annotated with AST-based `blame` and compare the results to ones computed by Sattler et al. [10] on line-based annotations. Again, we face the limitation of the performance overhead. Therefore, we only investigate the projects BROTLI, HTOP, LRZIP, and XZ.

5.3.1 *Results*

Figure 5.3 shows the results from running a central-code analysis on program files annotated with line-based `blame` (in red) and AST-based `blame` (in blue). The scatter plots compare the commit size (measured in the number of line insertions) to the node degree in the commit-interaction graph. A line between two dots shows that both dots represent the same commit. Results that are the same for both the line-based and the AST-based approach (in black) are primarily in the bottom left corner of the scatter plots. We observe one exception in the BROTLI project, where one common result is in the top left corner of the plot.

To further compare the two approaches, Figure 5.4 shows the difference between the two central-code analyses. For all analyzed projects, we observe many commits having a higher node degree with the AST-based approach compared to the line-based approach. In the

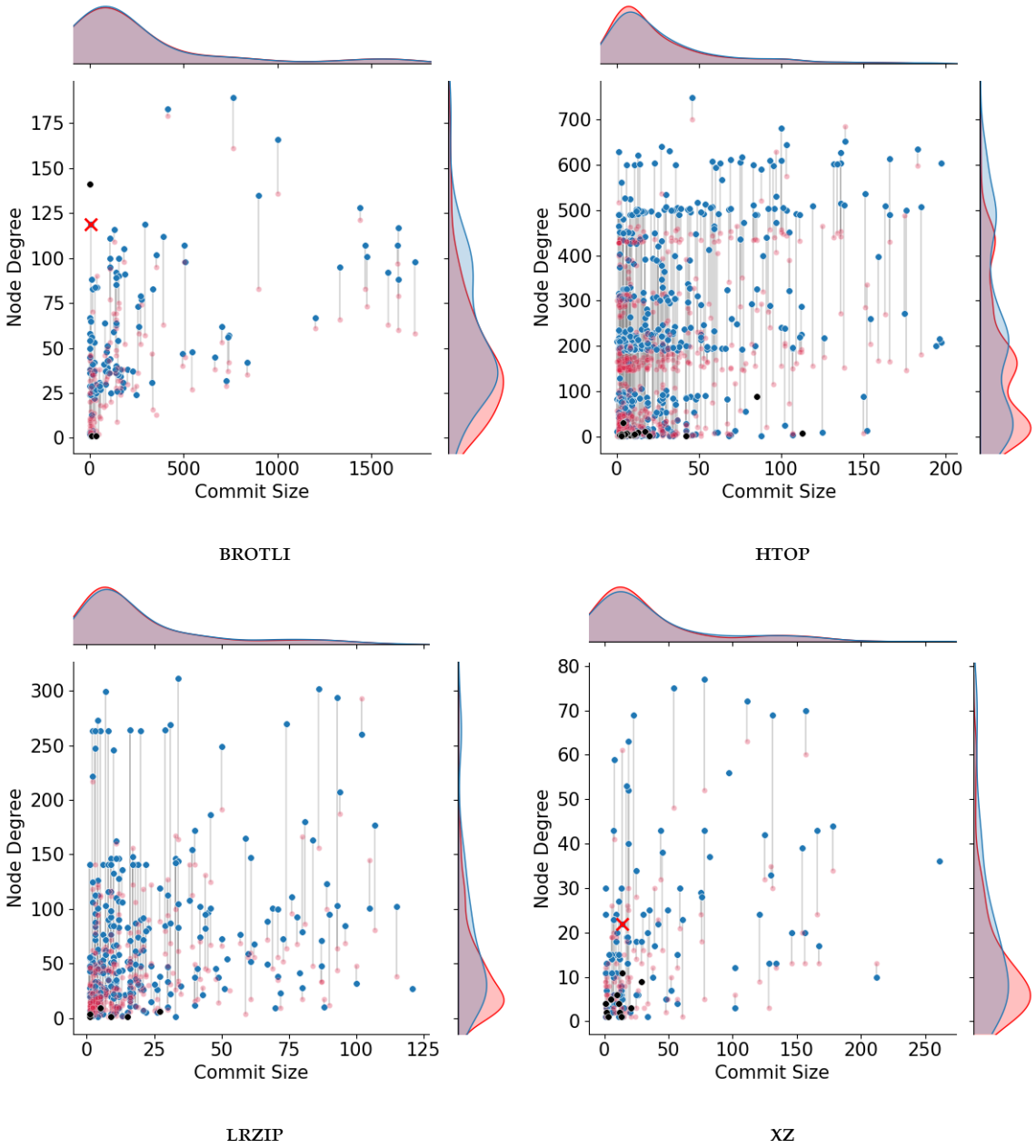

Figure 5.3: Results from central-code analysis if performed with AST-based annotations

plots for the BROTLI project, we highlight the commit `90fd2b60c` because it is a commit for which the AST-based result is significantly higher than the line-based one. In the plot for the xz project, we highlight the commit `7883d7353` because the value obtained with AST-based annotations is far lower than the one with line-based annotations. Overall, the difference for HTOP is again the most prominent, ranging between −456 and 550 interacting commits. We observe the minimal change in the project xz, ranging between −39 and 53 interacting commits. The amount of data points not changing by the transition from the line basis to the AST basis lies between 0.75% and 4.1%.

5.3.2 *Discussion*

The AST-based `blame` leads to a significant difference in `blame` annotations on an *IR* level. This is due to the fact, that we do not look at full lines but at individual characters. These different annotations lead to a changed commit-interaction graph. The central-code analysis, therefore, perceives different node degrees for different commits.

In general, a high node degree indicates that the commit introduces code central in the program execution. Subsequently, when we observe a higher node degree from the AST-based results compared to the line-based ones, it means that the commit has more influence on the program execution than previously assumed by the central-code analysis performed on line-based `blame` annotations. An example for this is the highlighted commit `90fd2b60c` in the BROTLI project. This shows that the AST-based `blame` can uncover previously undiscovered interactions between commits. With the line-based `blame`, the commit had a low node degree and did not stand out from other commits since it was in a cluster. By using AST-based `blame`, we uncover that even though it is a small commit, it has a big impact on the dependency structure of a program.

A lower node degree implies that the commit has less influence than previously assumed, meaning it is less important in the program execution. This is the case for commit `7883d7353` highlighted in the xz project. It is another indicator that the AST-based `blame` results help to identify influential commits more precisely.

To answer RQ2, we observe that the AST-based `blame` annotations have a big impact on the results from the central-code analysis. An indicator for this is the small percentage of data points not changing by the transition from the line basis to the AST basis. Therefore, between 95.9% and 99.25% of the commits have a different node degree than previously assumed when using the line-based `blame`. The central-code analysis is only one example of a commit-interaction analysis. For other commit-interaction analyses, we expect similar deviations. This observation raises a question for further work on whether the new results are actually more precise or just an amplification of the line-based results. This can not be answered without ensuring that all AST-based `blame` annotations are correct.

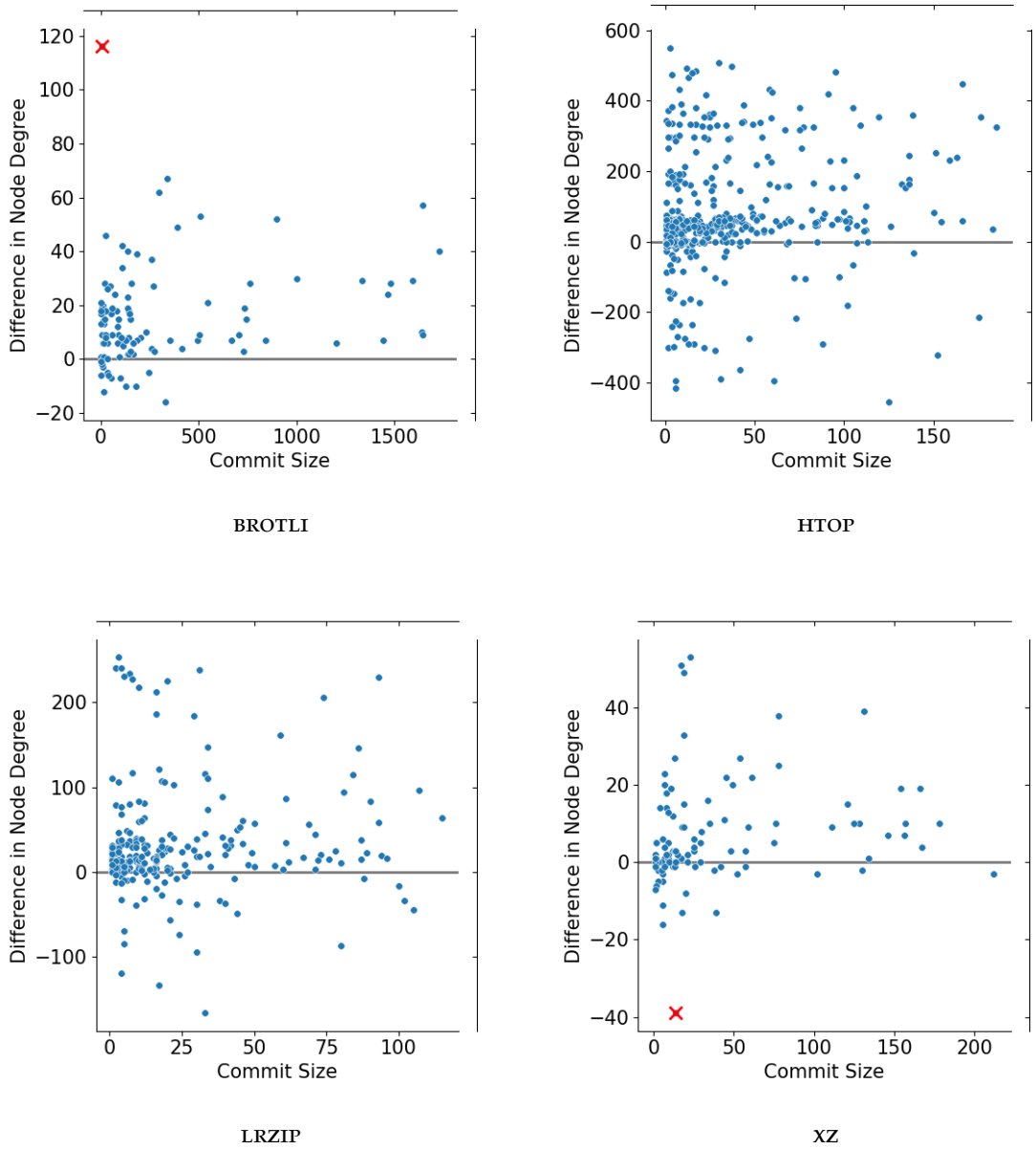

Figure 5.4: Change in results from central-code analysis if performed with AST-based annotations

6 RELATED WORK

The main reasons we want to add more fine-grained `git blame` information can be summed up into two categories: accurate source code authorship attribution and understanding source code evolution. The approach taken in this thesis improves the granularity of the `blame` information to help in these areas. There is a range of different approaches to improve these aspects. We start out with research that uses `blame` information and could profit from more fine-grained data. Then, we discuss a procedure that also tries to improve the `blame` information. Later in this chapter, we explore efforts to improve diff algorithms. Finally, we look at new ways of getting to know more information about authorship and code history.

6.1 USE CASES FOR BLAME INFORMATION

The `blame` information is necessary for many different analyses. Sattler et al. [10] annotate this information onto *IR* instructions and use it to create commit-interaction graphs. With the help of these graphs, they answer questions like which commits affect central code and how authors interact within a software project. Another research area that uses `blame` information deals with bug detection. The SZZ algorithm introduced by Śliwerski et al. [13] identifies commits with bug-fix-inducing changes. After finding a bug-fixing commit, it uses `blame` information to extract a set of possible bug-introducing commits.

6.2 IMPROVING BLAME DATA

German et al. [4] developed a tool called CREGIT that aims to provide token-based `blame` information. They achieved this by creating a so-called *repoView* of the analyzed repository. A *repoView* contains the same information that the original repository contains. The difference lies in the formatting. To be precise, each token in the *repoView* now gets a separate line. Such a transformation also changes the commit hashes. That is why German et al. [4] describe a procedure to correctly adjust the metadata of the repository with the new commit hashes. After the creation of the *repoView*, the `blame` functionality provided by *Git* can be used without any modification and we receive token-based `blame` information. In this thesis, we try to provide functionality that does not need any modification or duplication of the repository and can be directly used in the approach presented by Sattler et al. [10] for combining program analysis with repository information.

6.3 IMPROVING DIFF ALGORITHMS

Besides CREGIT, there have not been many papers that deal with adding fine-granularity to the `blame` functionality. However, there have been efforts to make the diff algorithms more fine-grained. Canfora et al. [1] introduce a language-independent diff algorithm which achieves that a line change is no longer interpreted as a line deletion followed by a line

insertion but can be distinguished from them. Fluri et al. [3] try to achieve something similar but on the AST layer by looking at and comparing AST nodes. These improvements are also usable in software development studies to improve the accuracy and gain better results.

6.4 OTHER APPROACHES

Meng et al. [7] try to gain more precise information about the author of certain code by adding new authorship models. The models introduced are *structural authorship*, which includes a subgraph containing all the commits involved in the history of the line, and *weighted authorship*, which assigns weights to authors based on their contribution to the line. This second metric is derived from the subgraph of the first metric.

Another problem that might arise in code authorship attribution can be anonymity. This is an area that is not addressed in this thesis. Dauber et al. [2] present an approach that uses random forests to identify the correct author.

Servant and Jones [11] address the issue of better understanding code development by introducing *history slicing*, which is an approach that aims to explore source-code history. This technique tracks the history of each line in the form of *history slices* which are all past revisions that modified the inspected line.

7 CONCLUDING REMARKS

This thesis addresses the line-based attribution problem we face when using `git blame` to annotate LLVM-*IR* instructions. The line-based `blame` annotates all instructions of one line with the same commit and that is the one that last modified the line, regardless of the size of the modification. When the commit only introduces a small change to a line, we lose the information that a previous commit introduced a subset of the corresponding instructions.

To overcome the line-based attribution problem, we introduce a character-based `blame` algorithm to annotate *IR* instructions with. The algorithm compares the line-based `blame` of a character with its parent commit to determine whether the character was already present in the parent commit. If that is not the case, we found the correct commit. Otherwise, we look further back into the repository history and investigate the line-based `blame` from the parent commit on backwards. This is repeated until the correct commit is found. The annotation of the *IR* instructions takes place during Clangs code generation phase. With our more fine-grained algorithm, we can assign different commit hashes to different *IR* instructions of the same line.

To decide whether we can benefit from the added fine-granularity, we construct small test repositories with changes to common programming constructs. By comparing the AST-based `blame` annotations to a manually created baseline, we observe an improvement in all cases except one, where the line-based result already equals the baseline. When further investigating the `blame` annotations of four real-world projects, we ascertain that an average of 17.6% of the AST-based annotations differ from the line-based ones. These different annotations lead to different results when performing a central-code analysis on real-world projects. Almost all commits have a different node degree, when we run the central-code analysis on program files with AST-based `blame` annotations compared to the line-based ones. The majority of the commits have a higher node degree than previously assumed. We also observe many cases where the node degree is significantly lower than assumed using line-based `blame`. Subsequently, we can more precisely identify the impact specific commits have on the program execution. We conclude that the character-based `blame` leads us to more precise annotations and results of commit-interaction analyses.

While our results indicate a substantial improvement over traditional line-based blame information, there are several limitations the character-based `blame` approach faces. In Section 3.3, we discuss that the character basis limits us when it comes to commits introducing name shortenings. This could be overcome by adding information about the start and end position of the tokens to the `blame` computation and would further improve the precision of the annotated commit hashes.

Another problem is the performance of the algorithm. This became especially obvious during the evaluation of RQ1.2 and RQ2, when annotating AST-based `blame` to the *IR* code of real-world projects. Iterating over characters and also using iteration in the character-based `blame` algorithm adds a significant overhead to the performance. This overhead

could be reduced by introducing a cache, so that the line-based `blame` does not have to be re-computed over and over again.

In the test repository regarding the namespace construct (Section 4.2.7), we noticed that it is not always clear which commit hash should be annotated to an instruction. Therefore, it would be interesting to introduce the ability to assign multiple hashes to an instruction. This also opens room for further commit-interaction analyses.

Another interesting opportunity for future research is the investigation of the different commit annotations between the line-based and the AST-based `blame` in more detail. We can examine, e. g., in how many cases a change in the annotation implies a change in the author and analyze the time difference between the newly assigned and the previously assigned commit.

In this thesis, we use a central-code analysis as a representative for commit-interaction analyses. It would also be interesting to see how the AST-based `blame` affects other commit-interaction analyses. Overall, the more fine-grained `blame` introduced in this thesis opens up many different starting points for future work.

APPENDIX

A.1 HISTORY OF TEST REPOSITORIES

A.1.1 *Case 1: Arithmetic Expression*

```
commit 2553b819d9434f3396727617438dc0d6ae39b056

    Add base implementation

 main.cpp | 11 ++++++++++

diff --git a/main.cpp b/main.cpp
--- /dev/null
+++ b/main.cpp
@@ -0,0 +1,11 @@
+#include <stdio.h>
+
+int calculate_something(int x) {
+  int result;
+  result = x + 42;
+  return result;
+}
+
+int main() {
+  printf("%d", calculate_something(4));
+}
```

```
commit 2b9a08d1adb63d2db9283811f79ac66e373ccfe9

    Subtract 1

 main.cpp | 2 +-

diff --git a/main.cpp b/main.cpp
--- a/main.cpp
+++ b/main.cpp
@@ -2,7 +2,7 @@

 int calculate_something(int x) {
   int result;
-  result = x + 42;
+  result = x + 42 - 1;
   return result;
 }
```

A.1.2 *Case 2: Conditional*

```
commit 8b2940f26d20e76ba7212935ccd507736fd8b368

    Add base implementation

 main.cpp | 12 ++++++++++++

diff --git a/main.cpp b/main.cpp
--- /dev/null
+++ b/main.cpp
@@ -0,0 +1,12 @@
+#include <stdio.h>
+
+bool conditional_function(int x, int y) {
+  if (x < y) {
+    return true;
+  }
+  return false;
+}
+
+int main() {
+  return 0;
+}
```

```
commit 73928b0410585375c7bdff62f985d376dec8ea22

    Adjust condition

 main.cpp | 2 +-

diff --git a/main.cpp b/main.cpp
--- a/main.cpp
+++ b/main.cpp
@@ -1,7 +1,7 @@
 #include <stdio.h>

 bool conditional_function(int x, int y) {
-  if (x < y) {
+  if (x <= y) {
     return true;
   }
   return false;
```

```
commit 990d0863b3c34b70d05dd4d0ecaed39f26a2d11b

    Add function conditional_declaration

 main.cpp | 7 +++++++

diff --git a/main.cpp b/main.cpp
--- a/main.cpp
+++ b/main.cpp
@@ -1,5 +1,12 @@
 #include <stdio.h>

+int conditional_declaration(int x) {
+  if (int y = x * 2) {
+    return y;
+  }
+  return 0;
+}
+
 bool conditional_function(int x, int y) {
   if (x <= y) {
     return true;
```

```
commit 2f5e3f926af735d3732495c1e0abd1cfe79d8366

    Change variable name of conditional decl

 main.cpp | 4 ++--

diff --git a/main.cpp b/main.cpp
--- a/main.cpp
+++ b/main.cpp
@@ -1,8 +1,8 @@
 #include <stdio.h>

 int conditional_declaration(int x) {
-  if (int y = x * 2) {
-    return y;
+  if (int result = x * 2) {
+    return result;
   }
   return 0;
 }
```

```
commit 54f96c7a33246c45cc0e6eddccee2bba73322bd1

    Change type of conditional decl

 main.cpp | 4 ++--

diff --git a/main.cpp b/main.cpp
--- a/main.cpp
+++ b/main.cpp
@@ -1,7 +1,7 @@
 #include <stdio.h>

-int conditional_declaration(int x) {
-  if (int result = x * 2) {
+float conditional_declaration(int x) {
+  if (float result = x * 2) {
     return result;
   }
   return 0;
```

A.1.3 *Case 3: Class*

```
commit dea47927754a6bb34009063a4cb893bc91004b2a

    Add base implementation

 main.cpp | 17 ++++++++++++++++

diff --git a/main.cpp b/main.cpp
--- /dev/null
+++ b/main.cpp
@@ -0,0 +1,17 @@
+#include <stdio.h>
+
+class Foo {
+  int i;
+public:
+  Foo(int i) : i(i) {};
+
+  int getVal() { return i; };
+  void setVal() { i = 42; };
+};
+
+int main() {
+  Foo fooA = Foo(21);
+  Foo fooB = Foo(42);
+  fooA.setVal();
+  return 0;
+}
```

```
commit 3255463271ac4799da8d07590bfefa6c0e99398a

    Rename class attribute i

 main.cpp | 8 ++++----

diff --git a/main.cpp b/main.cpp
--- a/main.cpp
+++ b/main.cpp
@@ -1,12 +1,12 @@
 #include <stdio.h>

 class Foo {
-  int i;
+  int value;
 public:
-  Foo(int i) : i(i) {};
+  Foo(int i) : value(i) {};

-  int getVal() { return i; };
-  void setVal() { i = 42; };
+  int getVal() { return value; };
+  void setVal() { value = 42; };
 };

 int main() {
```

```
commit 58459cfd97f9c6a8713a5c691fc36aaf0fe9fa67

    Rename parameter of constructor

 main.cpp | 2 +-

diff --git a/main.cpp b/main.cpp
--- a/main.cpp
+++ b/main.cpp
@@ -3,7 +3,7 @@
 class Foo {
   int value;
 public:
-  Foo(int i) : value(i) {};
+  Foo(int param) : value(param) {};

   int getVal() { return value; };
   void setVal() { value = 42; };
```

```
commit 7837d2aeec0f1bc2acca95184586cb2253689ad9

    Add body to constructor

 main.cpp | 4 +++-

diff --git a/main.cpp b/main.cpp
--- a/main.cpp
+++ b/main.cpp
@@ -3,7 +3,9 @@
 class Foo {
   int value;
 public:
-  Foo(int param) : value(param) {};
+  Foo(int param) : value(param) {
+    setVal();
+  };

   int getVal() { return value; };
   void setVal() { value = 42; };
```

```
commit e4d76dc0c7193cfa394e41acda03628cc23a7a48

    Change called function

 main.cpp | 2 +-

diff --git a/main.cpp b/main.cpp
--- a/main.cpp
+++ b/main.cpp
@@ -14,6 +14,6 @@ public:
 int main() {
   Foo fooA = Foo(21);
   Foo fooB = Foo(42);
-  fooA.setVal();
+  fooA.getVal();
   return 0;
 }
```

```
commit a965214eed5910b381524686c7d022f17b04cf7b

    Change object in function call

 main.cpp | 2 +-

diff --git a/main.cpp b/main.cpp
--- a/main.cpp
+++ b/main.cpp
@@ -14,6 +14,6 @@ public:
 int main() {
   Foo fooA = Foo(21);
   Foo fooB = Foo(42);
-  fooA.getVal();
+  fooB.getVal();
   return 0;
 }
```

```
commit b75ec269d73327f7b8f503026ae4146788b3ff72

    Make fooB a pointer

 main.cpp | 4 ++--

diff --git a/main.cpp b/main.cpp
--- a/main.cpp
+++ b/main.cpp
@@ -13,7 +13,7 @@ public:

 int main() {
   Foo fooA = Foo(21);
-  Foo fooB = Foo(42);
-  fooB.getVal();
+  Foo* fooB = new Foo(42);
+  fooB->getVal();
   return 0;
 }
```

```
commit 7e56aa94a76be6a0245368cb2bc48eb9158b83c8

    Introduce direct initialization

 main.cpp | 2 +-

diff --git a/main.cpp b/main.cpp
--- a/main.cpp
+++ b/main.cpp
@@ -12,7 +12,7 @@ public:
 };

 int main() {
-  Foo fooA = Foo(21);
+  Foo fooA(21);
   Foo* fooB = new Foo(42);
   fooB->getVal();
   return 0;
```

```
commit 41f1a96b84ab6e8158d4a0ca11babcc734e7e6bd

    Add variable with unary operator

 main.cpp | 1 +

diff --git a/main.cpp b/main.cpp
--- a/main.cpp
+++ b/main.cpp
@@ -13,6 +13,7 @@ public:

 int main() {
   Foo fooA(21);
+  Foo* fooA2 = &fooA;
   Foo* fooB = new Foo(42);
   fooB->getVal();
   return 0;
```

```
commit 865ea5dafee6f695e624b6c0079c211ad20d2bc7

    Change unary operator

 main.cpp | 4 ++--

diff --git a/main.cpp b/main.cpp
--- a/main.cpp
+++ b/main.cpp
@@ -12,8 +12,8 @@ public:
 };

 int main() {
-  Foo fooA(21);
-  Foo* fooA2 = &fooA;
+  Foo* fooA = new Foo(21);
+  Foo fooA2 = *fooA;
   Foo* fooB = new Foo(42);
   fooB->getVal();
   return 0;
```

A.1.4 *Case 4: FunctionCall*

```
commit 1de301ecb7d85789569a96aa77506d49962b1b94

    Add base implementation

 main.cpp | 5 +++++

diff --git a/main.cpp b/main.cpp
--- /dev/null
+++ b/main.cpp
@@ -0,0 +1,5 @@
+#include <stdio.h>
+
+int main() {
+  return 0;
+}
```

```
commit 3dfd6067d07f3cd29c538930b0238c08c74bfa89

    Add some_function and function call

 main.cpp | 5 +++++

diff --git a/main.cpp b/main.cpp
--- a/main.cpp
+++ b/main.cpp
@@ -1,5 +1,10 @@
 #include <stdio.h>

+int some_function(int x, bool y) {
+  return x;
+}
+
 int main() {
+  some_function(4, true);
   return 0;
 }
```

```
commit 3f90e82f596ceaca5f8815959a7cb1dbd341d85b

    Change call parameters

 main.cpp | 2 +-

diff --git a/main.cpp b/main.cpp
--- a/main.cpp
+++ b/main.cpp
@@ -5,6 +5,6 @@ int some_function(int x, bool y) {
 }

 int main() {
-  some_function(4, true);
+  some_function(4, false);
   return 0;
 }
```

A.1.5 *Case 5: FunctionDeclaration*

```
commit 765e3daaf901f82ec4f3fc645ae109b6e9e249c8

    Add base implementation

 main.cpp | 5 +++++

diff --git a/main.cpp b/main.cpp
--- /dev/null
+++ b/main.cpp
@@ -0,0 +1,5 @@
+#include <stdio.h>
+
+int main() {
+  return 0;
+}
```

```
commit a34c3dda0f5ed7a206c147a2490be6278242fac7

    Add some_function and function call

 main.cpp | 4 ++++

diff --git a/main.cpp b/main.cpp
--- a/main.cpp
+++ b/main.cpp
@@ -1,5 +1,9 @@
 #include <stdio.h>

+int some_function(int x) {
+  return x;
+}
+
 int main() {
   return 0;
 }
```

```
commit f3215786caabbe376c3fd27eb2a59e5eea14b2cc

    Add parameter to some_function

 main.cpp | 2 +-

diff --git a/main.cpp b/main.cpp
--- a/main.cpp
+++ b/main.cpp
@@ -1,6 +1,6 @@
 #include <stdio.h>

-int some_function(int x) {
+int some_function(int x, bool y) {
    return x;
 }
```

```
commit 6329be0233252da141399cb22ac0ac2af760b19c

    Change type of parameter

 main.cpp | 2 +-

diff --git a/main.cpp b/main.cpp
--- a/main.cpp
+++ b/main.cpp
@@ -1,6 +1,6 @@
 #include <stdio.h>

-int some_function(int x, bool y) {
+int some_function(int x, char y) {
    return x;
 }
```

```
commit ac8f8b4a0066d0086772fda69f5a0515f406b0ad

    Change name of parameter

 main.cpp | 4 ++--

diff --git a/main.cpp b/main.cpp
--- a/main.cpp
+++ b/main.cpp
@@ -1,7 +1,7 @@
 #include <stdio.h>

-int some_function(int x, char y) {
-  return x;
+int some_function(int result, char y) {
+  return result;
 }

 int main() {
```

A.1.6 *Case 6: Loop*

```
commit 9ab9c4366af9b628393332e57541ced8bcd3a304

    Add base implementation

 main.cpp | 5 +++++

diff --git a/main.cpp b/main.cpp
--- /dev/null
+++ b/main.cpp
@@ -0,0 +1,5 @@
+#include <stdio.h>
+
+int main() {
+  return 0;
+}
```

```
commit 46cfe6f1abd8aceeb4a16eabb324b3bef44a1200

    Add function loop_function

 main.cpp | 8 ++++++++

diff --git a/main.cpp b/main.cpp
--- a/main.cpp
+++ b/main.cpp
@@ -1,5 +1,13 @@
 #include <stdio.h>

+int loop_function() {
+  int result = 0;
+  for (int i = 0; i < 4; i++) {
+    result += i;
+  }
+  return result;
+}
+
 int main() {
   return 0;
 }
```

```
commit 963ab7f6e6730e6fbf8c3c4718ad1e5f3dbc87c7

    Change name of loop variable

 main.cpp | 4 ++--

diff --git a/main.cpp b/main.cpp
--- a/main.cpp
+++ b/main.cpp
@@ -2,8 +2,8 @@

 int loop_function() {
   int result = 0;
-  for (int i = 0; i < 4; i++) {
-    result += i;
+  for (int count = 0; count < 4; count++) {
+    result += count;
   }
   return result;
 }
```

```
commit 6968583cef191220de68d58cff255d1ac8a46bbd

    Add function enhanced_loop

 main.cpp | 9 +++++++++

diff --git a/main.cpp b/main.cpp
--- a/main.cpp
+++ b/main.cpp
@@ -1,5 +1,14 @@
 #include <stdio.h>

+int enhanced_loop() {
+  int array[] = {1, 2, 3, 4};
+  int result = 0;
+  for (auto n : array) {
+    result += n;
+  }
+  return result;
+}
+
 int loop_function() {
   int result = 0;
   for (int count = 0; count < 4; count++) {
```

```
commit 63bb631181eee27b5f89759998fdd52d0fa6265d

    Change auto to specific type

 main.cpp | 2 +-

diff --git a/main.cpp b/main.cpp
--- a/main.cpp
+++ b/main.cpp
@@ -3,7 +3,7 @@
 int enhanced_loop() {
   int array[] = {1, 2, 3, 4};
   int result = 0;
-  for (auto n : array) {
+  for (int n : array) {
     result += n;
   }
   return result;
```

```
commit d58b05e5bcb21c8d89d35c078e318822fe1a2d48

    Change name of array variable

 main.cpp | 4 ++--

diff --git a/main.cpp b/main.cpp
--- a/main.cpp
+++ b/main.cpp
@@ -1,9 +1,9 @@
 #include <stdio.h>

 int enhanced_loop() {
-  int array[] = {1, 2, 3, 4};
+  int numbers[] = {1, 2, 3, 4};
   int result = 0;
-  for (int n : array) {
+  for (int n : numbers) {
     result += n;
   }
   return result;
```

```
commit 2b4f207d0d0890f8ba689e375152ce29c75185fc

    Change name of enhanced loop variable

 main.cpp | 4 ++--

diff --git a/main.cpp b/main.cpp
--- a/main.cpp
+++ b/main.cpp
@@ -3,8 +3,8 @@
 int enhanced_loop() {
   int numbers[] = {1, 2, 3, 4};
   int result = 0;
-  for (int n : numbers) {
-    result += n;
+  for (int element : numbers) {
+    result += element;
   }
   return result;
 }
```

A.1.7 *Case 7: Namespace*

```
commit 87f116f8de58520da4eddb00350b89068bb98398

    Add base implementation

 main.cpp | 9 +++++++++

diff --git a/main.cpp b/main.cpp
--- /dev/null
+++ b/main.cpp
@@ -0,0 +1,9 @@
+#include <stdio.h>
+
+namespace my_namespace {
+  int x = 42;
+}
+
+int main() {
+  printf("%d", my_namespace::x);
+}
```

```
commit beba849763593b915152d64f5ec8a482bfcc0f06

    Introduce using keyword

 main.cpp | 4 +++-

diff --git a/main.cpp b/main.cpp
--- a/main.cpp
+++ b/main.cpp
@@ -4,6 +4,8 @@ namespace my_namespace {
   int x = 42;
 }

+using namespace my_namespace;
+
 int main() {
-  printf("%d", my_namespace::x);
+  printf("%d", x);
 }
```

BIBLIOGRAPHY

[1] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. "Tracking your changes: A language-independent approach." In: *IEEE software* 26.1 (2008), pp. 50–57.

[2] Edwin Dauber, Aylin Caliskan, Richard Harang, and Rachel Greenstadt. "Git blame who? stylistic authorship attribution of small, incomplete source code fragments." In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. 2018, pp. 356–357.

[3] Beat Fluri, Michael Wursch, Martin PInzger, and Harald Gall. "Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction." In: *IEEE Transactions on Software Engineering* 33.11 (2007), pp. 725–743.

[4] Daniel M. German, Bram Adams, and Kate Stewart. "Cregit: Token-Level Blame Information in Git Version Control Repositories." In: *Empirical Software Engineering* 24.4 (2019), 2725–2763.

[5] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. "A Systematic Literature Review on Fault Prediction Performance in Software Engineering." In: *IEEE Transactions on Software Engineering* 38.6 (2012), pp. 1276–1304.

[6] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Jr. Whitehead. "Automatic Identification of Bug-Introducing Changes." In: *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. 2006, pp. 81–90.

[7] Xiaozhu Meng, Barton P Miller, William R Williams, and Andrew R Bernat. "Mining software repositories for accurate authorship." In: *2013 IEEE international conference on software maintenance*. IEEE. 2013, pp. 250–259.

[8] Eugene W Myers. "An O (ND) difference algorithm and its variations." In: *Algorithmica* 1.1-4 (1986), pp. 251–266.

[9] Yusuf Sulistyo Nugroho, Hideaki Hata, and Kenichi Matsumoto. "How different are different diff algorithms in Git? Use–histogram for code changes." In: *Empirical Software Engineering* 25 (2020), pp. 790–823.

[10] Florian Sattler, Sebastian Böhm, Philipp Dominik Schubert, Norbert Siegmund, and Sven Apel. "SEAL: Integrating Program Analysis and Repository Mining." In: *ACM Trans. Softw. Eng. Methodol.* (2023).

[11] Francisco Servant and James A Jones. "History Slicing: Assisting Code-Evolution Tasks." In: *ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE)*. Cary, NC, 2012, 43:1–43:11.

[12] Tamanna Siddiqui and Ausaf Ahmad. "Data mining tools and techniques for mining software repositories: A systematic review." In: *Big Data Analytics: Proceedings of CSI 2015* (2018), pp. 717–726.

[13] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. "When do changes induce fixes?" In: *ACM sigsoft software engineering notes* 30.4 (2005), pp. 1–5.