Leibniz Universität Hannover
Institut für Systems Engineering
Fachgebiet System- und Rechnerarchitektur

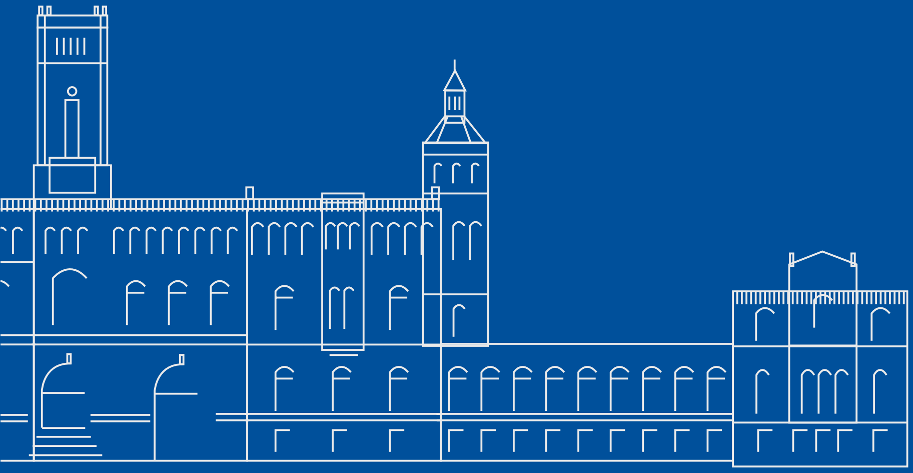
Leibniz
Universität
Hannover

Maximilian Michael Werner

Efficient Change Impact Quantification by Global AST Hashing

Masterarbeit im Fach Technische Informatik

6. Dezember 2021



Leibniz Universität Hannover
Institut für Systems Engineering
Fachgebiet System und Rechnerarchitektur
Appelstr. 4 · 30167 Hannover · Germany

www.sra.uni-hannover.de

Efficient Change Impact Quantification by Global AST Hashing

Masterarbeit im Fach Technische Informatik

vorgelegt von

Maximilian Michael Werner

angefertigt am

Institut für Systems Engineering
Fachgebiet System- und Rechnerarchitektur

Fakultät für Elektrotechnik und Informatik
Leibniz Universität Hannover

Erstprüfer:	**Prof. Dr.-Ing. habil. Daniel Lohmann**
Zweitprüfer:	**Prof. Dr.-Ing. Bernardo Wagner**
Betreuer:	**M.Sc. Tobias Landsberg**

Beginn der Arbeit:	**5. Mai 2021**
Abgabe der Arbeit:	**5. November 2021**

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties. I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Maximilian Michael Werner)
Hannover, 6. Dezember 2021

ABSTRACT

The increasing complexity of software systems results in changes to these systems often unknowingly causing unintended side effects. Particularly in the case of large open source projects with developers new to the project, this leads to an increased review effort and a higher risk for errors. For this reason, it is desired that possible side effects of a change can be detected and quantified as early as possible.

With the *c*Hash approach, there is a novel way to detect semantic changes to program code and distinguish them from semantically irrelevant changes based on abstract syntax tree (AST) hashing. In conjunction with the concept of the global reference graph (GRG), the dependencies within a program can be modelled completely. Using the GRG, change impact analysis (CIA) can be performed to estimate the impact of a change on the rest of the program. To better estimate the severity of a change, various metrics are defined to quantify change impact. The main computational effort in the presented CIA is performed by *c*Hash, and thus by the compiler. The additional overhead of computing the change metrics turns out to be low, making this approach a very efficient means of impact analysis.

First the metrics were examined for their expressiveness using the development history of the open source projects QEMU, CPython, OpenSSL and Lua. It was then investigated whether there is a correlation between the size of a change and the social interaction around that change in an open source project. The evidenced discrepancy in these two measurements suggests that developers are often unaware of the possible impact of a change or do not discuss it properly. A comparison with a third party metric showed that the change metrics presented can certainly match established solutions.

KURZFASSUNG

Die steigende Komplexität von Softwaresystemen führt dazu, dass Änderungen an diesen Systemen oftmals unbewusst unerwünschte Seiteneffekte auslösen. Insbesondere bei großen Open-Source Projekten mit Entwicklern, die noch nicht so lange mit dem Projekt vertraut sind, führt das zu einem erhöhten Review-Aufwand und einer höheren Fehlergefahr. Aus diesem Grund ist es wünschenswert, die möglichen Seiteneffekte einer Änderung frühzeitig erkennen und quantifizieren zu können.

Mit dem *c*Hash-Ansatz gibt es eine neuartige Möglichkeit, semantische Änderungen am Programmcode zu erkennen und von semantisch irrelevanten Änderungen zu unterscheiden, die auf dem Hashing des Abstrakten Syntaxbaums (AST) aufbaut. Zusammen mit dem Konzept des Globalen Referenzgraphen können die Abhängigkeiten innerhalb eines Programms vollständig abgebildet werden. Mit dem Globalen Referenzgraphen können Änderungsanalysen durchgeführt werden, die die Auswirkungen einer Änderung auf das komplette Programm abschätzen. Um die Größe einer Änderung besser einschätzen zu können, werden verschiedene Metriken zum Quantifizieren der Änderungsauswirkungen definiert. Der hauptsächliche Rechenaufwand bei der vorgestellten Änderungsanalyse wird von *c*Hash, und damit vom Compiler getragen. Der zusätzliche Aufwand durch das Berechnen der Änderungsmetriken fällt gering aus, wodurch mit diesem Ansatz eine sehr effiziente Art der Änderungsanalyse gefunden wurde.

Die Metriken wurden zunächst anhand der Entwicklungsgeschichte der Open-Source Projekte QEMU, CPython, OpenSSL und Lua auf ihre Aussagekraft untersucht. Anschließend wurde überprüft, ob ein Zusammenhang zwischen der Größe einer Änderung und der sozialen Interaktion um diese Änderung in einem Open-Source Projekt besteht. Die nachgewiesene Diskrepanz dieser beiden Messungen legt nahe, dass Entwicklern oftmals das Ausmaß einer Änderung nicht bewusst ist oder diese nicht ausreichend diskutiert wird. Ein Vergleich mit der Metrik eines Drittanbieters ergab, dass die vorgestellten Änderungsmetriken durchaus mit etablierten Lösungen mithalten können.

CONTENTS

1 INTRODUCTION

Software maintenance is a crucial part of the software development lifecycle: A software product rarely reaches a finished and final state. After their release, most projects are constantly evolved and adapted to account for changing requirements, environments and new or old security vulnerabilities. Consider, for example, the Linux kernel: After its initial release in 1991[1] it grew from roughly 8.400 lines of code to now over 22 million lines of code.[2] Through this massive increase in size and complexity, a developer can hardly always fully understand the possible side effects and consequences a new change to the software could have. Therefore, to assist the software development and review process, an entire research area has been established around change impact analysis. Since impact analysis was first defined by Arnold and Bohner in 1993, a plethora of methods have been explored with varying degrees of precision and complexity. The approaches can be static or dynamic and range from call graph analysis over program slicing to ripple effect analysis [Li+13].

A common denominator across all those approaches is that they result in impact sets. Those describe what parts of a program will possibly be affected by a change. During software development and review, those impact sets can be used to assess the possible consequences of a change. However, invoking an impact analysis program and inspecting the impact sets manually for each change can quickly be seen as an additional burden to a developer, rather than an opportunity. Therefore, an impact analysis approach that cleanly integrates with a projects build process, which is invoked anyway during development, is desirable. The interpretation of the impact analysis can be greatly simplified for the developer if the possible effects of a change are presented in a report using metrics that quantify different aspects of a change. Impact analysis can be much more valuable if its results lead to error prevention early in the development process, rather than in change review.

In this thesis, a new change impact analysis approach based on Dietrich et al.'s *c*Hash compiler plugin for abstract syntax tree (AST) hashing and Landsberg et al.'s concept of the global reference graph is presented. The global reference graph is a unique representation of the semantic structure of a program. Using graph-theoretic techniques, the impact sets of a change can be identified and the extent of a change can be quantified. Because the approach is based on the non-intrusive C compiler plugin *c*Hash, this new kind of impact analysis can be easily integrated into existing build systems. The main effort of impact analysis with *c*Hash is offloaded to the compiler, hence this approach adds only little overhead to the usual build process.

This thesis is structured as follows: In Chapter 2, an overview of the fundamentals is provided. First, impact analysis is defined in more detail and alternative approaches are presented. Then, the necessary terms and algorithms from graph theory are introduced. With the help of these principles, the approaches of *c*Hash and the global reference graph are explained. This is followed by an overview

[1]Original Linux kernel release: `https://mirrors.edge.kernel.org/pub/linux/kernel/Historic/`

[2]Lines of code, ignoring empty lines and comments, counted with `cloc`: `https://github.com/AlDanial/cloc`

of related work. Chapter 3 describes in detail, how the impact sets are generated leveraging *c*Hash and the global reference graph. Furthermore, various metrics for quantifying change impact are introduced. Those metrics are analyzed and compared to other software measurements in Chapter 4. The results of this thesis are discussed in Chapter 5.

FUNDAMENTALS 2

This chapter explains the fundamentals of change impact analysis (CIA). A brief overview of graph theory and related algorithms is given, which helps understand some concepts in this thesis. Subsequently, the cHash approach and its connection to CIA are discussed. This is followed by the explanation of Global AST Hashing, the foundation for this thesis. Finally, related work is addressed.

2.1 Change Impact Analysis

Software maintenance makes up for a big portion in the software development life cycle. Maintenance modifications can be classified as either corrective, preventive, adaptive or perfective [ISO06] and each of these changes could degrade the software's quality if not carefully reviewed (e.g. by introducing bugs, slowdowns or other unwanted side effects). Those changes either stem from changing requirements or become necessary due to malfunctions in the software. Schneidewind outlined several problems in software maintenance [Sch87]: It is difficult to detect whether a change in code will affect the system. A given change may have unexpected side effects.

The research field of software change impact analysis investigates those questions from different angles. Several techniques have been introduced to explore parts of the software that need to be changed for a proposed modification. Other CIA methods reveal components that have changed after modifying the software. CIA can therefore be used before and after applying changes. It helps with understanding software and tracing the effects of change to it [Li+13].

2.1.1 Definition

A widely accepted definition for CIA was formulated by Arnold and Bohner: "Impact analysis (IA) is the activity of identifying what to modify to accomplish a change, or of identifying the potential consequences of a change." [AB93]. Software change impact analysis is a part of the maintenance process and should contain, among others, the following steps [ISO06]:

- Identify ripple effects.
- Determine the level of test and evaluation required.
- Estimate the size and magnitude of the modification.
- Consider the development history of a program.

CIA methods can be classified by the levels of project artifacts they take into consideration for their analysis. Traceability-based approaches try to link high-level documents (such as requirements, use-cases) to their resulting objects (such as source code) and help with understanding the relationship between these artifacts [DLFO08]. These approaches are primarily concerned with the first step in the CIA process and result in a change set (Figure 2.1). The change set, also known as starting impact set (SIS), consists of all objects of the software that need to be changed in order to implement a new requirement.

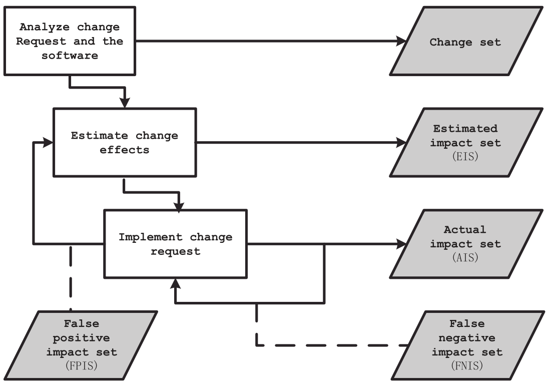

Analyze change Request and the software
Change set
Estimate change effects
Estimated impact set (EIS)
Implement change request
Actual impact set (AIS)
False positive impact set (FPIS)
False negative impact set (FNIS)

Figure 2.1 – Change Impact Analysis process [Li+13]

Contrary to this, dependence-based approaches analyze relations between software components on the same level of abstraction. More recent examples mostly focus on source code analysis [Li+13]. On this level of abstraction, the CIA methods can directly estimate the impact of modifications on the final product (second step in Figure 2.1).

The various CIA approaches have in common that they estimate an impact set (IS) from the changes to be made or already implemented [Li+13]. Impact sets contain all elements of a program/ project (files, functions, variables, …) that are (possibly) subject to change. The SIS, as already mentioned, contains all initially affected elements. The estimated impact set (EIS) is the result of CIA approaches and contains objects that are thought to be indirectly affected after applying the SIS [AB93]. After analyzing a change request, it is implemented. This may result in an actual impact set (AIS) that differs from the EIS (third step in Figure 2.1).

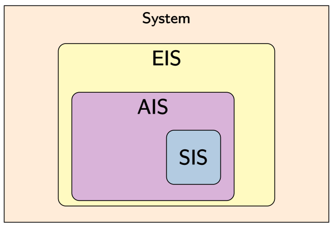


Figure 2.2 – Exemplary system with the impact sets of a proposed change

If the CIA approach overestimates the EIS, the complement of AIS in EIS is called false positive impact set (FPIS). The set of objects which appear in the AIS but are missing in the EIS is called false negative impact set (FNIS) [Li+13]. It is safe to assume that the starting impact set always appears in both the EIS and AIS. Figure 2.2 shows a program with a highly impactful proposed change (EIS is half the systems size). The applied impact analysis presents an EIS that slightly deviates from the AIS. An ideal CIA process would always estimate an IS that matches the actual impact set [AB93].

In practice we would accept an approach as in Figure 2.2. The FPIS is nontrivial, but it is only slightly larger than the AIS. More importantly, the FNIS is empty and does not lead to programming errors based on wrong assumptions about the AIS. The exemplary CIA is thus *complete*, but not necessarily *sound* [RY20].

To compare the accuracy of the different CIA approaches, *precision* and *recall*, measures originally used in information retrieval, were introduced [Hat+08]. Both rely on the previously defined false positive or false negative IS and measure how severely a CIA approach misestimates the IS.

Precision indicates which fraction of the EIS was actually modified and appears in AIS. It is defined as in Equation (2.1). The fewer false positives (small FPIS) found, the higher the precision.

$$P = \frac{|EIS \cap AIS|}{|EIS|} \tag{2.1}$$

Recall indicates the proportion of modified objects in AIS that is also found in EIS. It is defined as in Equation (2.2). The fewer objects in the FNIS, the higher the recall.

$$R = \frac{|EIS \cap AIS|}{|AIS|} \tag{2.2}$$

For both measures, a value close to 1 is desirable to have high confidence in a CIA approach. *Precision* and *recall* are ways to quantify *soundness* and *completeness* and therefore useful for comparing CIA approaches.

2.1.2 Common Methods

Change impact analysis methods can roughly be grouped into static and dynamic analysis and each approach may be attributed with their own strengths and weaknesses [Li+13]. Traditional static analysis builds upon dependency tracing in a dependency graph. Given a graph that correctly represents all dependencies in a program, a static CIA method will estimate a *complete* IS. However, not each dependency in the graph may actually change, so a static approach can produce a significant FPIS. Dynamic approaches are based upon data that is collected during program execution. When running a program, there is no guarantee that every possible code path will be taken. It is thus not possible to observe every change, but every change that is observed is guaranteed to belong to the AIS. Dynamic CIA can be considered *sound*, but the FNIS may not be empty [Hat+08]. In the following, some common static approaches are presented, since the approach in this thesis can also be classified as static.

Ripple effect is a measure of structural complexity. It can be used for understanding the relationships in a module or program but also as a way to measure impact. The ripple effect of a variable is computed by tracing its impact on the rest of a program. For each variable and each line it appears on the right-hand side of an assignment it is tracked, which other variables it affects. From those traces a $0-1$ matrix is constructed, which shows the dependencies between variables inside a module. Subsequently, a $0-1$ matrix is generated in which the effect of each variable within a module on other modules is captured. This way of conducting IA is *complete*, but can become computationally very expensive for large modules and programs [Bla08].

Program slicing describes the process of removing parts of a program until it only contains everything that is necessary for performing one specific behaviour of the initial program [Ton03]. If the slicing process is started at a specific variable or function, the remaining program can be seen as the IS. It is not unusual for a single slice to contain up to 30 % of the initial program code. This causes slicing approaches to have poor *precision*. Various approaches are being taken to make slicing more accurate, including dynamic approaches and manual corrections. However, these approaches are relatively complex [GHR09].

Recent CIA techniques focus on *mining software repositories*. These approaches allow tracing of co-change between files and/or software objects. Dependencies between artifacts that are hard to observe with conventional static analysis can be revealed with these approaches [Li+13].

Static call graph analysis is one of the fundamental approaches to CIA. A static call graph is a directed Graph $G(F, C)$, where F is a set of nodes representing functions in the program. Function calls are represented by the set $C \subseteq F \times F$. The calling relationships are computed from the source code of a program, in contrast to the *dynamic call graph*, where function calls are traced throughout program execution. Given changes to the functions $\{f_{c0}, \dots, f_{cn}\} \in F$, the IS is computed by considering the transitive closure (see Section 2.2.2) over $\{f_{c0}, \dots, f_{cn}\}$ and f_{root} in G [BBSY05]. Call graphs have several disadvantages in relation to CIA. Not each return value of a function call directly impacts its direct or indirect callers. Therefore, it can be assumed that this approach will yield a large FPIS. Even worse, CIA with call graphs can lead to significant FNIS, since changes to structures, global variables and other non-function program members are not considered at all. To mitigate some of these issues, Badri et al. propose *control call graphs*, a combination of *control flow graphs* and *call graphs*. The more precise information from the control call graph results in the FPIS being smaller [BBSY05].

2.2 Graph Theory

The principles of graph theory play an important role in this thesis and graphs will be employed for multiple purposes. This thesis' title implies that abstract syntax trees (ASTs) will be the subject of research. Those ASTs are composed into a global reference graph (see Section 2.4), which is technically a *directed graph*. The data from different Git repositories is used for evaluation. A Git project's development history is represented as a *directed acyclic graph*. In the following section, a brief overview over those data structures and algorithms to examine them is given.

2.2.1 Definitions

A *Graph* is a mathematical structure composed of *nodes* (or *vertices*) and *edges* (or *lines*). It is formally noted as a pair $G = (V, E)$ of sets. The nodes v are contained in V, the edges are contained as pairs $e = (v_x, v_y)$ (or $e = v_x v_y$) in E. In the following, nodes and edges may be referred to as $v \in G$ and $e \in G$ (instead of $v \in V(G)$ and $e \in E(G)$), to reduce verbosity [Die17].

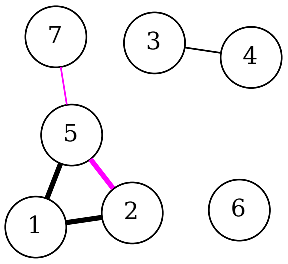
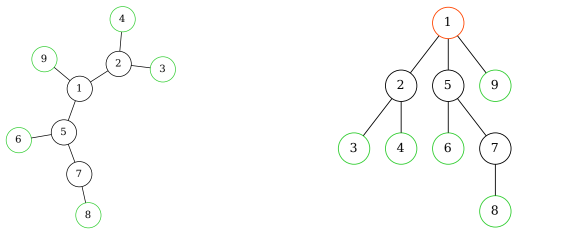


Figure 2.3 – A Graph with the vertices $V = \{1, \dots, 7\}$ and edges $E = \{(1, 2), \dots\}$ [Die17]

Figure 2.3 depicts a simple graph, yet containing interesting properties. It is nontrivial, since its *order* $|G|$, the number of nodes, is greater than one.

This graph contains multiple *paths* $P(V', E')$, which are *subgraphs* containing nodes $V' = \{v_0, v_1, \dots, v_x\} \in V$ and edges $E' = \{v_0 v_1, v_1 v_2, \dots, v_{x-1} v_x\} \in E$. The *length* of a path is the number of its edges. An example would be the path $P_{magenta} = \{\{2, 5, 7\}, \{(2, 5), (5, 7)\}\}$ of length 2, with the edges highlighted in magenta. The graph also contains a *circle* $C_{bold} = \{\{1, 2, 5\}, \{(1, 2), (2, 5), (5, 1)\}\}$, which is a path P extended by an edge $e = v_x v_0$ connecting the *endvertices*. It is highlighted with bold edges. The graph in Figure 2.3 is not *connected*, since there does not exist a path between each pair of nodes in the graph. However, the graph contains three *components* $V(C_1) = \{1, 2, 5, 7\}$, $V(C_2) = \{3, 4\}$ and $V(C_3) = \{6\}$. Components are maximal connected subgraphs [Die17].

A graph is called a *forest*, if it contains no cycles and is therefore *acyclic*. A connected forest is called a *tree*. In a tree, each node v with a *degree* of 1, the number of edges connected to v, is called a *leaf*. In a *rooted tree*, a special node is called *root* [Die17]. Figure 2.4 shows a simple tree and the same tree in its rooted representation. The leaves are colored green, the root is colored red.



(a) A simple tree **(b)** The same tree with root node 1

Figure 2.4 – Tree and rooted tree

A prominent example for trees in computer science is the abstract syntax tree (AST), an intermediate source code representation in compilers. The AST represents the hierarchical syntactic structure of the source code. Its root is an expression, the children can be subexpressions and their operands [ASU86].

A *directed graph*, short *digraph*, is a graph with oriented edges. The edges, sometimes also called *arcs*, are now defined as $e = (u, v)$, where the first node u is the *tail* and the second node v is the *head*. Paths and cycles are defined similarly as in undirected graphs. A path must however follow the direction of the edges. If a digraph contains no cycles, it is called directed acyclic graph (DAG) [BJG09]. Considering a *directed rooted tree*, the root always has an *in-degree* (or *out-degree*) of 0 [Die17].

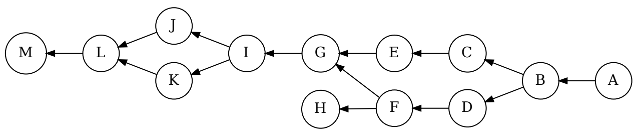


Figure 2.5 – A simple directed acyclic graph

Figure 2.5 shows a DAG. It contains multiple paths, for example from A to M. This path does not exist in the reversed direction, since there are no edges oriented in this direction. In an undirected graph, the nodes $\{A, B, C, D, E, F, G\}$ and $\{I, J, K, L\}$ would form cycles. This is not the case for this directed graph, since the participating edges are not oriented in a circular form.

2.2.2 Algorithms

In the following sections, some standard graph-related algorithms will be mentioned and used. These will not be explained or discussed in detail, however, a brief overview is considered necessary. The algorithms will not be implemented as part of this thesis, since there are libraries providing those features in an efficient manner.[3]

[3]The NetworkX Python package: `https://networkx.org/`

Transitive Closure

A directed graph G is *transitive*, if for each pair of edges xy and yz, the edge xz is also in G [BJG09].

The *transitive closure* $TC(G)$ is a transitive digraph containing the same nodes as the original graph G. Each edge uv in $TC(G)$ exists if and only if there is a path from u to v in G [BJG09]. The transitive closure can be computed in reasonable time using Warren and Warshall's algorithm [War75].

Figure 2.6 shows a simple digraph G (solid black edges). The transitive closure $TC(G)$ consists of all edges, including the dashed grey edges.

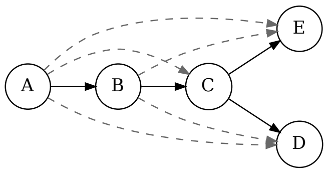


Figure 2.6 – A directed graph and its transitive closure

Finding Shortest Paths

The *distance* $dist(x, y)$ of nodes x to y in a graph G is the minimum length of an (x, y)-path, if such a path exists. Otherwise, $dist(x, y) = 0$ [BJG09]. Figure 2.7 shows a digraph with multiple paths from node A to D. Path $ABCD$ has length 3, the two shortest paths ABD and ACD have length 2. Therefore, $dist(A, D) = 2$.

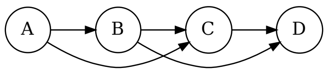


Figure 2.7 – A digraph with multiple paths from node A to D

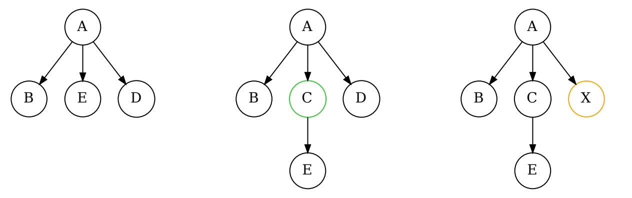
Numerous algorithms have been proposed to find the shortest paths and distance between two nodes in a (di-)graph, such as Dijkstra's, the Bellman-Ford-Moore and the Floyd-Warshall Algorithm. In the case of an unweighted directed graph, using a modified breadth-first search (BFS) has proven to be sufficient [BJG09].

Tree and Graph Edit Distance

For many application domains, such as bioinformatics, pattern recognition or database research, it is interesting to quantify the difference between two given graphs or trees [PA11]. This measure of (dis-)similarity is known as tree edit distance (TED) or in the generalized form as graph edit distance (GED).

Pawlik and Augsten define TED $\delta(F, G)$ (or GED) as the minimum-cost sequence of node edit operations transforming graph F into G. The three basic operations consist of *deleting*, *adding* and *renaming* a node. Each of those operations can be weighted with an individual cost [PA11]. Other authors also include edge edit operations in the cost computation, which is especially interesting with weighted edges [Ser19].

Figure 2.8 shows a tree and two steps of transformation. Given the edit costs $c_{add} = c_{delete} = c_{rename} = 1$, each transformation $a \rightarrow b$ and $b \rightarrow c$ has an edit distance of 1. The first cost is caused by inserting node C, the second is caused by renaming D into X.



(a) A simple tree **(b)** Insert node *C* **(c)** Rename *D* into *X*

Figure 2.8 – Example trees and edit operations, from left to right [PA11]

Both GED and TED are expensive to compute. Fischer et al. state that an optimal GED algorithm runs in exponential time in regards to the size of the graphs and therefore propose a suboptimal approximation, which still requires quadratic time [FRB17].

The optimal TED algorithm has a cubic runtime in regards to the size of the trees. Pawlik and Augsten have therefore presented the robust tree edit distance (RTED) algorithm and more recently the all path tree edit distance (AP-TED) algorithm, which select the optimal strategy for computing TED based on the shape of the input trees [PA11; PA15; PA16].

2.3 cHash

A common problem across software projects with compiled languages are long build times. Incremental changes to the source code lead to frequent compiler invocations, not all of which are necessary and therefore considered *redundant builds*. Several approaches have been proposed for performing *incremental builds*: Instead of recompiling each source file in a project, only those affected by a change will be rebuilt, thus reducing redundant builds. In some sense, this problem is closely related to change impact analysis: A build system must identify the SIS and from there determine the EIS, i.e., all the objects that need to be rebuilt [AB93]. Such a build system is, however, subject to certain constraints:

- It must be *complete*, so no necessary recompilation will be missed.
- The FPIS should be as small as possible, otherwise too much time will be spent on the incremental builds, which diminishes the build system's utility.
- Determining the EIS has to be faster than rebuilding an object file. Otherwise, rebuilding every object file for each iteration would be cheaper, which defeats the purpose of a build system.

The tool *Make* is, despite its age, still among the most commonly used build systems.[4] The user-provided `makefile` describes how to build a program (or any other kind of software product) and what intermediate object files are required to build it. It also describes any dependencies between the individual objects. Make derives its SIS based on timestamps. If a source file has been touched more recently than its resulting object file, it belongs to the SIS. Using the dependencies described in the `makefile`, Make works out the EIS. All objects contained in the EIS will be rebuilt according to the user-provided rules [MO05]. This way of calculating the impact set is cheap and easy to implement. However, since only `touch`ing a file is enough to trigger recompilations, when nothing actually changed, a large FPIS has to be expected when using Make.[5] This assumption is supported

[4]GNU-make: `https://www.gnu.org/software/make/`
[5]Unix-touch: `https://man7.org/linux/man-pages/man1/touch.1.html`

by recent studies which show that up to 97 % of rebuilds are redundant when using classical Make [Zha+15]. In addition, the EIS generated by Make cannot be guaranteed to be *complete*. Make derives its impact set from the `makefile`, which in some cases is handwritten and may be subject to oversights.

A more sophisticated tool for building the EIS, and therefore the list of objects to be rebuilt, can be found in Ccache [RT]. This tool intervenes the compilation process after the preprocessing stage. A hash is calculated over the preprocessed source code and the compilation is aborted if the same hash can already be found in the tools internal cache. This can even speed up clean builds significantly, since previous compilation results can be pulled from the cache. However, this approach still produces a FPIS: Changes to syntactic and semantic constructs that do not result in a different compiled object (e.g., additional declarations) will still lead to unnecessary recompilation [Die+17].

Dietrich et al. propose *c*Hash, an approach that operates similar to Ccache, but mitigates some of its imprecisions. Unlike Ccache, *c*Hash calculates a hash over the AST. Therefore, the compilation process is interrupted at a later stage, after preprocessing, parsing and semantic analysis. This additional overhead must be worthwhile in regards to the spared redundant recompilations, otherwise *c*Hash would not be ideal for selecting recompilation candidates [Die+17].

```c
struct unused {
  struct unused * next;
};

struct obj {};

struct refcount {
  int counter;
  struct obj * ptr;
};

int
inc(struct refcount * e)
{
  e->counter += 1;
  return e->counter;
}
```

Listing 2.1 – C source code [Die+17]

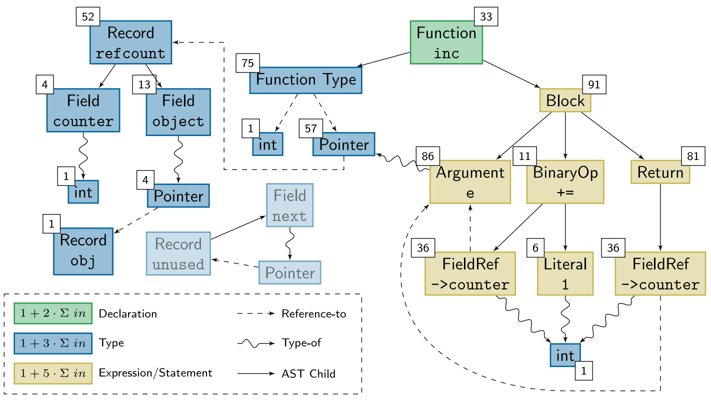


Figure 2.9 – The *c*Hash approach: Each node recursively gets assigned an individual hash [Die+17]

Listing 2.1 shows an excerpt of a C source module, Figure 2.9 shows its corresponding AST. It consists of one function and three record definitions. The AST is no longer depicted as a real tree, since *reference-to* and *type-of* relationships have been added as edges. Through those cross-tree references, cyclic structures can appear in the AST, which now actually is a digraph [Die+17].

To calculate a hash over the whole AST, and therefore a hash for the corresponding source file, *c*Hash explores the graph with a depth-first search (DFS). Each node's hash is a combined value of its own class and the hash values of all of its referenced nodes. Since a node can be visited multiple times with this approach, the following rules have been introduced: 1) If a node's hash was calculated before, reuse it. 2) If a node is encountered while currently visiting it, use a surrogate hash value for it instead. A textual representation of the type name is used as the surrogate value [Die+17].

By exploring the AST with a DFS, components of the graph that are not linked to the root node will be excluded from the hash calculation. Considering the example in Figure 2.9, the record `unused` appears in an isolated component and is therefore not covered in the top-level AST hash.

This behaviour is desired and an improvement over Ccache. Changes to source code, which do not modify the AST and therefore do not impact the resulting binary, are excluded from recompilation.

In practice, *c*Hash has originally been implemented as a plugin for the `clang` C compiler and later been adapted to `gcc`.[6] For hashing, the efficient MurMur3 function is used. Dietrich et al. state that *c*Hash's semantic approach is at least 30.19 % more precise in detecting unnecessary recompilations than Ccache [Die+17].

The *c*Hash compiler plugin presented by Dietrich et al. is a central pillar for this thesis. It is used to generate the local hashes for the global reference graph, which is further discussed in Sections 2.4 and 3.1.

2.4 Global AST Hashing

In modern software development, regression testing has become a standard practice to provide reliable systems. By immediately testing even small changes to the software, unwanted behaviour can be spotted early and fixed easily. However, always running the full test suite after each change is computationally expensive and quickly becomes infeasible for large projects. Regression-test selection (RTS) describes the approach of only selecting tests that are required to cover a given change. Effectively, RTS boils down to impact analysis: Changes to the software-under-test (SUT) are the SIS, tests selected for reexecution are the EIS. With TASTING, an approach for RTS based on AST hashing has been presented [LDL21].

In TASTING, the semantic hashing of functions and ASTs first presented in *c*Hash has been reused to fingerprint each test-execution binary. (Test-)programs consist of multiple object files linked into an executable binary. A compiler computes the AST for a single compilation unit, which is compiled into an object file. To compute the semantic fingerprint for a whole program, Landsberg et al. introduce the concept of a global reference graph (GRG), spanning over multiple source files [LDL21].

This additional graph contains all relevant functions and global variables of a program, but omits the detailed subtree of each expression. It can be generated with minimal intervention into a projects normal build process: 1) The compiler is invoked with additional command line arguments pointing to the *c*Hash plugin. The plugin places an AST with semantic hashes into a separate section in the resulting object file. 2) The linker is instructed to output a cross-reference table (CRT), which describes the relationship between symbols across object file boundaries. 3) An additional program is invoked, which extracts the fingerprinted AST from all involved object files of a program binary and combines them to a global reference graph using the relationships found in the CRT [LDL21].

Since *c*Hash only computes hashes for a single source file, the following terms were introduced: *Local hashes* are the fingerprints generated by *c*Hash. *Global hashes* are added to propagate the node's dependencies across source file boundaries into the root node of the global reference graph (i.e., the `main`-function).

$$H(f_0) = h(f_0) \oplus \underbrace{\left(\bigoplus_{f \in l(f_0) \setminus \mathrm{SC}(f_0)} H(f) \right)}_{\textit{child functions}} \oplus \underbrace{\left(\bigoplus_{f \in \mathrm{SC}(f_0) \setminus \{f_0\}} h(f) \right)}_{\textit{recursive group}} \tag{2.3}$$

The global hash $H(f)$ (see Equation (2.3)) for a GRG node f (a function or global variable) is defined recursively using the following functions: The local hash function $h(f)$ is defined in *c*Hash.

[6]Integration of AST Hashing into the GCC compiler: `https://www.sra.uni-hannover.de/Theses/2017/BA-cHash-gcc-plugin.html`

The link function $l(f)$ gives all successor nodes to f. The function $\mathrm{SC}(f)$ calculates the strongly connected subgraph for f. A node's global hash is the hashed concatenation $\oplus$ of its own local hash and the global hashes of all its *child functions*. Possible cycles in the reference graph caused by recursion have to be treated with special care: If a child function f_c is contained in the same strongly connected subgraph $\mathrm{SC}(f_0)$ as the current node f_0, exclude it from the *child functions* part of $H(f_0)$ and instead concatenate its local hash $h(f_c)$ as part of the *recursive group*. This avoids cyclic dependencies during the global hash calculation [LDL21].

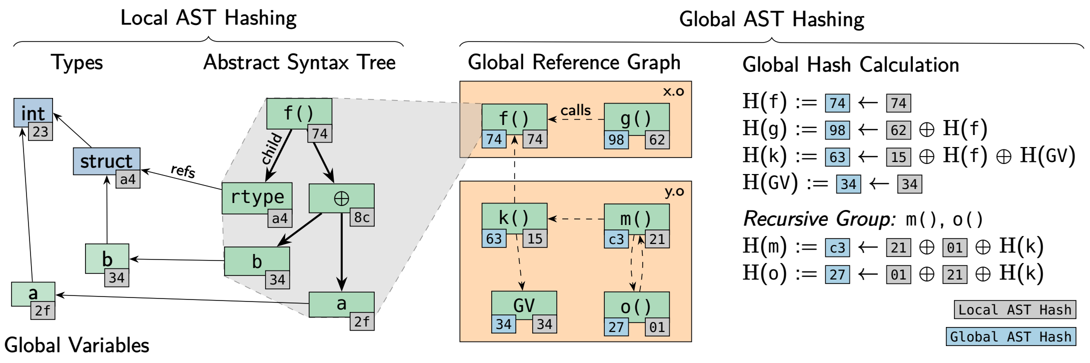


Figure 2.10 – Overview of the TASTING Approach for Global AST Hash Calculation [LDL21]

Figure 2.10 gives an overview over the global AST hash calculation. The left side depicts an excerpt from an AST, similar to Figure 2.9, with the local hashes $\boxed{\texttt{lh}}$ calculated as described for *c*Hash. On the right, an excerpt of the global reference graph can be seen. The object-spanning relationship between function `k()` and `f()` has been deduced from the CRT. The global hashes $\boxed{\texttt{gh}}$ are calculated as described by Equation (2.3) [LDL21].

Assuming there are no hash-collisions, the TASTING approach can be used to predict if two programs could behave differently: If the global hash of the programs root nodes (`main`) differ, the underlying global reference graph and ASTs must be at least slightly different. Even though calculating the fingerprints for each test adds overhead to the testing process, TASTING could drastically reduce time spent on tests in two of three examined projects. As in *c*Hash, the MurMur3 function was used for all hashing purposes. Since TASTING currently builds upon the *c*Hash compiler plugin, it is only available for projects written in the C programming language [LDL21]. However, the concepts presented are flexible enough to be ported to similar languages. The current implementation of the TASTING Approach for Global AST Hash Calculation, `chash-global`, is written in the Rust programming language.[7]

The concept of the global reference graph and `chash-global` presented by Landsberg et al. will be used in this thesis to deduce the impact sets of a change. In contrast to the possibly uncomplete call graph analysis, the GRG as a unique representation of a program can be used to implement complete IA.

[7]Rust: `https://www.rust-lang.org/`

2.5 Related Work

Most research in the field of IA is concerned with estimating an impact set for a proposed software change (see Section 2.1.2). Nevertheless, some approaches for quantifying change impact have been presented.

Harding proposes Line Impact, a metric to measure the *cognitive load* required to author a Git commit.[8] It effectively weights the relevant lines of code that are modified in any given commit. It has been shown that LineImpact correlates between 26 % and 63 % with difficulty estimations assigned to tasks that are completed in a commit [Har21].

Füracker implemented a tool for global hash calculation that is very similar to the TASTING approach, the latter being published four years later. The tool has been used to characterize the development history of the Lua interpreter.[9] He showed that in an average commit around 4.9 % of local hashes have changed. A comparison to call graph analysis shows the great similarity between both approaches. However, the global reference graph also reflects changes to global variables and record data types [Fü17].

Some effort has been put into finding a relationship between change discussion and their corresponding commits. Baysal and Malton use natural language processing (NLP) to find correlation between source code change history and social interaction surrounding those changes. They conducted their research on two different projects, Apache Ant[10] and LSEdit[11], with a very wide range of releases and messages examined. One of their findings was that for Apache Ant, the number of messages correlates with the extent of changes, whereas for LSEdit it does not correlate well.

Wu studied the concept of punctuated software evolution in open source projects. He concludes, that software systems evolve through long periods of small incremental change and short periods of sudden large changes [Wu06].

Chilowicz et al. and Feng et al. both use AST fingerprinting to detect code duplicates. They present their approaches as superior over text-, token- or syntax-based duplicate detection. Their goal is to detect plagiarism or common code modification antipatterns like intra-project copy-pasting [CDR09; FCX13].

Sager et al. present *Coogle* (Code Google), a tool for detecting similar classes and methods within a codebase. They use different tree similarity measures to detect code structures with a similar AST representation. Among all the tree similarity measures studied, tree edit distance performed best, but at the cost of high computational complexity [Sag+06].

[8]Git SCM: `http://git-scm.com/`
[9]Lua—the programming language: `https://www.lua.org/`
[10]Apache Ant: `https://ant.apache.org/`
[11]LSEdit: The Graphical Landscape Editor: `https://www.swag.uwaterloo.ca/lsedit/index.html`

ARCHITECTURE 3

It was already suggested in Chapter 1 that it is desirable to be able to perform change impact analysis efficiently and automatically. Think of large open source projects that are maintained and developed by many people distributed all around the world, like the Linux kernel or the GNU Project. The open development model of these projects allows anyone to submit improvements. However, few of the contributors have a fully comprehensive understanding of the interdependencies of the software to which they are submitting changes. It would therefore be valuable to have a tool that can estimate the potential impact of a change. The impact of a change can then be quantified using various metrics, allowing the developer to easily judge the results of the impact analysis. Unintended side effects can thus be identified and eliminated early, before they lead to potential errors in the software.

In order to quantify the impact of a change, this chapter first describes how *c*Hash and the concept of the global reference graph are used to perform impact analysis. This is done by describing how the starting and estimated impact set are defined and derived from the GRG using local and global hashes. The impact sets are then used to obtain metrics for measuring the impact of a change.

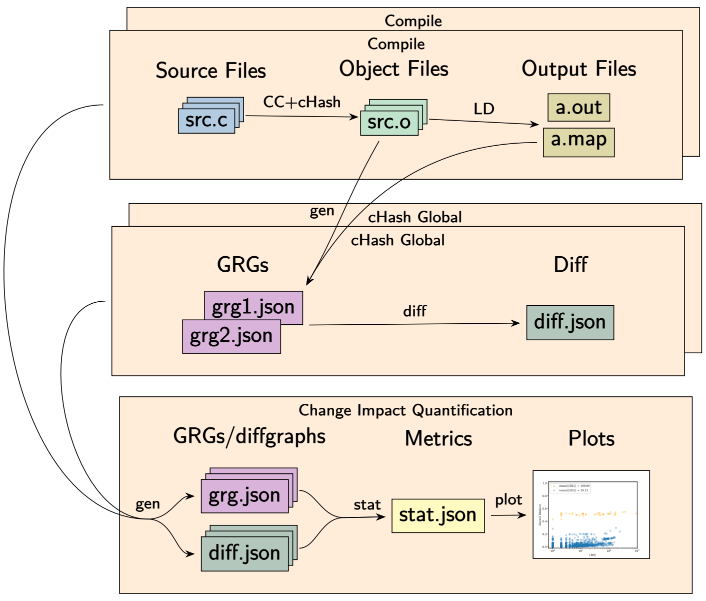


Figure 3.1 – Change Impact Analysis Workflow

Figure 3.1 shows the three-step workflow just described. First, the program versions before and after a change are compiled with the *c*Hash plugin. From this, the GRGs are constructed, which are used to derive the impact sets. Finally, various metrics are applied to the impact sets to quantify the impact of the change.

3.1 Defining the Impact Set

The presented CIA approach is situated in the third step of the change impact analysis process presented in Figure 2.1: The change is quantified after implementing a change request. Given a program P and an advanced version P', we want to calculate change metrics for the change $C = P \rightarrow P'$. Since determining the impact of change C based on program binaries for P and P' is not feasible, we rely on the global reference graph as a surrogate for the programs semantic structure (see Section 2.4). The GRG is a digraph $G = (V, E, w)$ with the nodes V representing the programs definitions (functions, global variables) and edges E representing their dependency relationship. Function $w(v) = (h(v), H(v))$ describes a mapping between nodes and their weights, in this case the local and global hash.

Using the local and global hash of each node, the impact sets of change C can be estimated. The starting impact set, as defined in Section 2.1.1, should contain all functions that are changed directly by the programmer. Therefore, if the local hash of node v has changed $(h(v_P) \neq h(v_{P'}))$, it counts towards the SIS. Nodes that have been added to or removed from P are also part of the SIS, which is denoted as the symmetric difference $\triangle$ of two sets. Equation (3.1) shows the definition for the SIS.

$$\text{SIS} = \underbrace{\{v \in \text{GRG}_P \cap \text{GRG}_{P'} | h_{\text{GRG}_P}(v) \neq h_{\text{GRG}_{P'}}(v)\}}_{\textit{local hash changed}} \cup \underbrace{(\text{GRG}_P \triangle \text{GRG}_{P'})}_{\textit{new or deleted}} \tag{3.1}$$

All nodes that could be affected by a change to a node in the SIS become part of the estimated impact set. Consider the example in Listings 3.1 and 3.2: Function `decide` is part of the SIS, since it has been changed to now always return `false`. Function `main`, even though it has not changed directly, becomes part of the EIS, since it depends on the behaviour of `decide`.

```
bool decide() { return true; }
                         /**/
int main() {
    if (decide()) do_this();
    else          do_that();
}
```

Listing 3.1 – Code example P

```
bool decide() { return false; }
                         /***/
int main() {
    if (decide()) do_this();
    else          do_that();
}
```

Listing 3.2 – Adapted code example P'

As defined in Equation (2.3), a node's global hash changes if its own or any of its successors local hash changes. Therefore, if the global hash of a node has changed, it is part of the estimated impact set. New or deleted nodes are also considered to be part of the EIS, since their global hash changes from $\varnothing \rightarrow H(v)$ or vice versa. The EIS is therefore per definition a superset of SIS. It can be formally described as in Equation (3.2). Since the *new or deleted* nodes are also part of the SIS, and the SIS is a subset of EIS, the symmetric difference of GRGs can be substituted by SIS.

$$\begin{aligned}\text{EIS} &= \underbrace{\{v \in \text{GRG}_P \cap \text{GRG}_{P'} | H_{\text{GRG}_P}(v) \neq H_{\text{GRG}_{P'}}(v)\}}_{\textit{global hash changed}} \cup \underbrace{(\text{GRG}_P \triangle \text{GRG}_{P'})}_{\textit{new or deleted}} \\ &= \{v \in \text{GRG}_P \cap \text{GRG}_{P'} | H_{\text{GRG}_P}(v) \neq H_{\text{GRG}_{P'}}(v)\} \cup \text{SIS}\end{aligned} \tag{3.2}$$

Conducting impact analysis based on the GRG can be classified as dependence-based CIA. The relation between source code definitions is analyzed, which are all located on the same level of abstraction [Li+13]. The estimated impact sets are assumed to be complete, which will not be formally proven. The estimation is based on *c*Hash and and the global reference graph presented in TASTING, for whose validity the respective authors have made strong arguments [Die+17; LDL21]. Note, that the impact sets are only defined by the nodes hashes, ignoring the edges. This is sufficient, since an additional outgoing edge from a node *n* implies that *n* has changed internally (i.e., additional function call), which is reflected by an updated local and global hash.

3.2 Constructing the Impact Set

As already stated, the starting and estimated impact sets will be generated by leveraging the concept of the global reference graph presented by Landsberg et al. Constructing the impact sets consists of two steps: 1) The GRG is built for two program versions P and P'. 2) The IS is generated by examining the GRGs. This section describes the process in detail.

3.2.1 Constructing the Global Reference Graph

In order to build the GRG with `chash-global`, the AST and local hashes for each translation unit need to be extracted. This information is generated by employing the *c*Hash approach. Each translation unit is compiled as usual, but the compiler is invoked with the *c*Hash-plugin. The AST, enriched by the local hashes, is placed into a new section in the compiled object file by the compiler plugin [LDL21]. This section is called `.gnu.lto_chash.ElementHashes` and is discarded during the linking process, resulting in a program binary with the same characteristics as if it had been built without *c*Hash. The AST with hashes is encoded in the JSON data interchange format, which is simple to parse for further inspection and transformation.[12] Currently, embedding AST hashes into object files is only supported by the *c*Hash plugin for the `clang` C compiler.[13] For each program binary to be produced, the linker is instructed to also output a `map`-file containing the cross-reference table, as described in Section 2.4.[14]

After compiling a project with *c*Hash, `chash-global` is instructed to build the GRG.[15] It is possible to define a `target` program, for which the GRG will be generated. Otherwise, a combined GRG is built for all `map` files. This would be the case if a project has multiple linking targets, for example for the main executable and multiple unit test programs. It is also possible to select an entry-point, the root node for the GRG. The default entry-point is `main`. Once the global reference graph is built, it is output in the JSON format for further inspection.

[12]JavaScript Object Notation (JSON): `https://www.json.org/json-en.html`

[13]Additional required `clang`-options for AST hash embedding: `-fplugin=path/to/chash-plugin -Xclang -plugin-arg-clang-hash -Xclang -generate-info`

[14]Additional required linker-options: `-Map,binary_name.map,--cref`

[15]Using the subcommand `chash-global gen`

```c
int val = 10;

int factorial(int n) {
  if (n == 0)
    return 1;
  else
    return n * factorial(n-1);
}

int main() {
  printf("%d\n", factorial(val));
}
```

Listing 3.3 – A simple C program computing the factorial of an integer

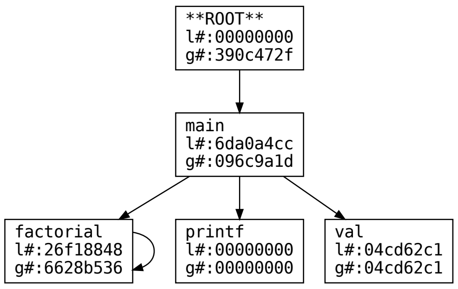


Figure 3.2 – The global reference graph for the factorial program

Listing 3.3 shows a small C program that calculates the factorial of an integer recursively and prints the result. The corresponding GRG with local (`l#`) and global (`g#`) node hashes can be seen in Figure 3.2. This example has been chosen to introduce some properties of `chash-global` that have not been discussed yet. The node `**ROOT**` is added to each GRG graph in order to have a defined root node. This is necessary, since the entry point to a program may not always be a function with the name `main`. It is also used in TASTING, where multiple test binaries are composed into a single GRG in order to reuse the global hashes of object files used in multiple test programs. The node `**ROOT**` is a synthetic node which always has a local hash zero. The function `printf` is part of the C standard library (libc), which is not compiled as part of the program. Therefore, `chash-global` assigns the local (and global) hash zero as a placeholder. The function `factorial` computes the factorial of an integer recursively. This recursion is reflected by the self-loop in the graph. Note, that the global hash of the self-referencing function `factorial` differs from its local hash, even though this can not be explained with Equation (2.3). This difference is caused by implementation details of `chash-global` and does not impact the outcome of the presented impact analysis approach: A node only appears in an impact set if its global hash has changed through an edit to the source code, a difference between the local and global hash of a node is not relevant for impact sets.

In the global reference graph, each node is named with a unique identifier so that it can be referenced in another GRG when comparing those graphs.[16] The identifier consists of the nodes symbol, which includes the name, type ((static) function or variable, synthetic node) and path to the source file the node is defined in. In addition to that, the identifier tracks the object file the node was found in and the target, for which the object file was built.

By computing the global reference graph in this way, all semantic changes to a program are signaled by a change to the root node's global hash. However, some changes have no significant impact on the programs behaviour. For example, many programs contain a version tag which is automatically computed at compile time and often depends on a current source code management (SCM) tag. The version tag is often represented as a global variable inside the GRG and a change to it causes all functions using it to change their global hash as well. Such nodes can be excluded from global hash computation by passing a list of ignored nodes to `chash-global`. Internally, these nodes' local hash will be replaced by zero.

In some cases, adding or removing a line from a source file can change the local hash of other, seemingly unrelated functions in the same file. The cause of this is that the affected functions use macros such as `assert`, which internally use the predefined macro `__LINE__`. In the preprocessing

[16]Not in Figure 3.2, for readability reasons.

stage, `__LINE__` is replaced by the line number it is placed on. When the line this macro is used on shifts up- or downwards, its value and therefore the local hash of the depending function changes. This macro is most frequently used for debugging and rarely has significant impact on a programs actual behaviour. It can therefore be overwritten with a constant value when the program is compiled for change impact analysis.[17]

3.2.2 Generating the Impact Set

Given two program versions P and P', the starting and estimated impact set for a change $C = P \rightarrow P'$ can now be calculated using the global reference graphs. For each node in both GRGs it is part of the SIS if its local hash has changed, and part of the EIS if its global hash has changed. Nodes that occur in only one of the graphs are part of both impact sets.

Generating the impact sets this way can be easily implemented with set operations. By using this approach, valuable information about the relationship between the impacted nodes is lost. However, information about the relationship between nodes will be useful for some of the metrics presented later. Therefore, other approaches to determining impact sets from GRGs will be explored.

Graph Edit Distance

Graph edit distance appears to be a well suited algorithm for finding the impact sets, enriched by the edit operations necessary to transform GRG_P into $\text{GRG}_{P'}$. Depending on the implementation chosen for GED, not only does it return the edit cost to transform the graphs, but also an edit path containing all edit operations (see Section 2.2.2). The implementation chosen for evaluation can be found in the NetworkX Python package.

In order to reduce computational effort, several constraints can be submitted to the algorithm: 1) A pair of root nodes (`**ROOT**`) that has to be matched across both graphs. 2) Functions for determining if two nodes should be matched. For this, the nodes unique identifier will be used.

While the GED algorithm works for simple test programs (such as in Listing 3.3), its exponential behaviour quickly proves impractical for mid- to large-scale software projects. A test run of the algorithm with two GRGs from the Lua interpreter, each with approximately 1020 nodes[18], could not be carried out completely, since the memory of the test machine (32 GiB) was not sufficient. Therefore, another approach to compute the impact sets must be found.

Differential Graph

GED algorithms are not optimal for determining the difference between two GRGs, since the algorithms were formulated generally for any kind of graph. To efficiently determine the difference between two GRGs, one can take advantage of the properties of the graphs: Each node has a unique identifier. If the global hash of an outermost node of a subgraph has not changed, the complete subgraph has not changed. The proposed method that exploits these properties is called differential graph (short: *diffgraph*) and has already been implemented in a draft version in `chash-global`.

Algorithm 3.1 shows how a differential graph for a change $C = P \rightarrow P'$ between two global reference graphs is computed. The first loop takes all nodes from P' and checks via the unique identifier if they also appear in P. If a node appears in both graphs, but its global hash has not changed, it will not be part of the diffgraph, since neither it nor any node in a subgraph of the current node has changed. If the nodes global hash has changed, it will be added to the diffgraph.

[17] Additional compiler option: `-D__LINE__`

[18] The smallest of the examined projects, see Chapter 4.

```
Require: Global reference graphs $\mathrm{GRG}_P$ and $\mathrm{GRG}_{P'}$
 1: $\mathrm{GRG}_{\mathrm{diff}} \leftarrow \{\emptyset, \emptyset\}$
 2: for all $v \in \mathrm{GRG}_{P'}$ do
 3:     $\mathrm{change}_v \leftarrow$ unchanged
 4:     if $v$ IN $\mathrm{GRG}_P$ then
 5:         if $H(v_{\mathrm{GRG}_P}) \neq H(v_{\mathrm{GRG}_{P'}})$ then
 6:             if $h(v_{\mathrm{GRG}_P}) \neq h(v_{\mathrm{GRG}_{P'}})$ then
 7:                 $\mathrm{change}_v \leftarrow$ changed
 8:             end if
 9:             $\mathrm{GRG}_{\mathrm{diff}} \leftarrow \mathrm{GRG}_{\mathrm{diff}} \cup \{v\}$
10:         end if
11:         $\mathrm{GRG}_P \leftarrow \mathrm{GRG}_P \setminus \{v\}$
12:     else
13:         $\mathrm{change}_v \leftarrow$ added
14:         $\mathrm{GRG}_{\mathrm{diff}} \leftarrow \mathrm{GRG}_{\mathrm{diff}} \cup \{v\}$
15:     end if
16: end for
17: for all $v \in \mathrm{GRG}_P$ do
18:     $\mathrm{change}_v \leftarrow$ removed
19:     $\mathrm{GRG}_{\mathrm{diff}} \leftarrow \mathrm{GRG}_{\mathrm{diff}} \cup \{v\}$
20: end for
21: for all $e(v_1, v_2) \in \mathrm{GRG}_P \cup \mathrm{GRG}_{P'}$ do
22:     if $\{v_1, v_2\}$ IN $\mathrm{GRG}_{\mathrm{diff}}$ then
23:         $\mathrm{GRG}_{\mathrm{diff}} \leftarrow \mathrm{GRG}_{\mathrm{diff}} \cup \{e\}$
24:     end if
25: end for
26: return $\mathrm{GRG}_{\mathrm{diff}}$
```

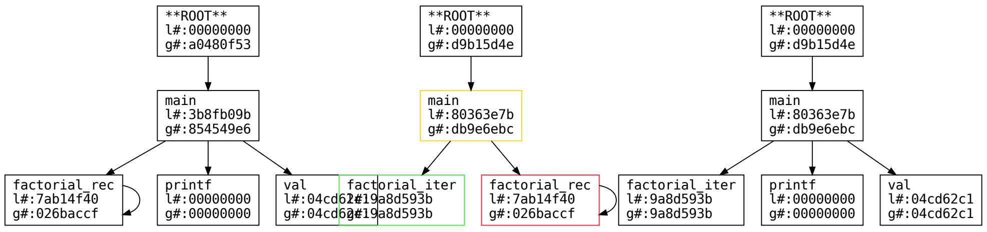
Algorithm 3.1 – Diffgraph algorithm

If its local hash has changed, it will be additionally marked as *changed*. Then, the node is removed from graph P. If a node cannot be found in GRG_P, it is marked as *added* and then also placed in the diffgraph.

The second loop adds all nodes to the diffgraph that were removed from P in the change C. Since in the first loop all non-unique nodes were removed from GRG_P, it now only contains nodes that do not exist in $\mathrm{GRG}_{P'}$. Those nodes are added to the diffgraph with the mark *removed*.

Finally, in the third loop all edges that exist in either GRG_P or $\mathrm{GRG}_{P'}$ are added to the diffgraph. This way, the relationship between nodes in both graphs is preserved. The entirety of the nodes in the diffgraph forms the EIS. All nodes marked as deleted, added or modified form the SIS.

Figure 3.3 gives an overview of the results of the diffgraph calculation. Figure 3.3a shows the GRG of the recursive factorial program from Listing 3.3, whereas Figure 3.3c shows another version of the program in which the factorial is determined iteratively rather than recursively. The diffgraph from both GRGs is shown in Figure 3.3b. It can be seen that the function `main` has been changed, because it now uses the iterative instead of the recursive factorial function. Therefore, the two functions are now shown as deleted and added, respectively. The other nodes (`val` and `printf`) are not part of the diffgraph and therefore also not part of the impact sets, because their local and global hash has not changed in the new version.



(a) GRG$_P$ for program P calculating factorial recursively **(b)** The diffgraph for change $C = P \rightarrow P'$ **(c)** GRG$_{P'}$ for program P' calculating factorial iteratively

Figure 3.3 – Two global reference graphs for different versions of a program computing the factorial of an integer and their corresponding diffgraph. In the diffgraph, nodes are colored as follows: added → green; deleted → red; changed → yellow. Nodes that are exclusively part of the EIS are colored black.

The diffgraph algorithm can be implemented in linear runtime: The GRGs are represented internally as a hashmap, which means that each access to a node or edge has constant runtime. This leaves the three sequential loops for runtime considerations, whose runtime depends linearly on the size of the input graphs. These properties make the diffgraph algorithm the preferred choice over GED.

Considerations on the Impact Sets

In this thesis, the diffgraph is used to determine the impact sets of a software change. This method can be implemented efficiently by building upon the results from the *c*Hash and TASTING papers. In the following section, the peculiarities of the impact sets resulting from this decision will be discussed.

By using local hashes for determining the SIS, this set itself is actually also only an estimate of the initial change set. This is caused by the properties of *c*Hash. Considering a change to the source code, the SIS is not always immediately visible: Preprocessor macros and definitions (`#define`) are not represented as nodes in the AST and GRG, since their textual replacement takes place before the syntactic analysis, where the AST is generated and *c*Hash intervenes. A single change to a macro can therefore cause multiple functions using this macro to change their hash, which leads to an unexpectedly large SIS. This was also described in Section 3.2.1, when the concerns related to the `__LINE__` macro were discussed.

Figure 3.3 illustrated another possible problem with the SIS. The function `factorial_iter` may already exist in the program code for program version P. However, since it is only called in the new version P', it appears in the related GRG for the first time. By only looking at the diffgraph, it may appear as though the code for this function was first added in the change C.

The EIS found in the diffgraph can be compared to the impact set found with call graph analysis. The resulting diffgraph contains all nodes that are contained in the transitive closure over all impacted nodes (the SIS) and the root node of the GRG. However, the diffgraph method takes all changes to the source code into consideration, that may change the behaviour of the examined program. Therefore, in contrast to call graph analysis, the approach can be considered complete.

The EIS contains all nodes that *could* be affected by a change, whereas the AIS contains all definitions that *will* be affected. To make a statement about the AIS, a more precise method than the one presented in this thesis would be required. Therefore, nothing will be said about the precision or recall of global reference graph analysis.

3.3 Program Wide Metrics

With the approach presented in Section 3.2 it is now possible to generate the impact set of a change as a side result of the compilation process. However, since such impact sets can quickly become incomprehensibly large, it is helpful to have metrics that can be used to evaluate the changes. The following section presents simple yet meaningful ways to rank said impact sets.

3.3.1 Jaccard Distance

The *Jaccard index* is a measure of similarity between two sets. It is defined as the intersection over the union of two sets A and B, see Equation (3.3) [Jac12].

$$J_{idx}(A,B) = \frac{|A \cap B|}{|A \cup B|} \tag{3.3}$$

Analogously, *Jaccard distance* is defined as a measure of dissimilarity between two sets. It can be described using the Jaccard index as in Equation (3.4).

$$J_{dist}(A,B) = 1 - J_{idx}(A,B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|} \tag{3.4}$$

In practice, sets A and B will contain the global hashes of the global reference graphs of two program versions. The Jaccard distance can be used to relate the EIS of a change to the GRGs of two program versions. Furthermore, this metric can be used to quickly compare the impact of multiple changes. Jaccard distance (and index) are defined for values between zero and one. In this application, a Jaccard distance close to zero indicates a small possible effect on the program behavior, whereas a distance of one indicates a possibly strong change. The Jaccard distance is therefore the normalized size of the estimated impact set.

Not only can this metric be used to quantify the size and impact of changes. If a series of unrelated changes to a single program continuously scores high Jaccard distances, this may indicate that the examined program has a tight internal coupling: In a tightly coupled program, the software modules tend to have many dependencies, i.e. many edges in the GRG. When compared to a more loosely coupled program, those additional edges in the GRG cause a large EIS and therefore a high Jaccard distance more often.

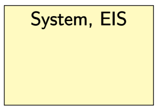
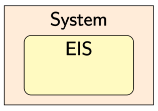
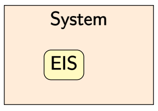
Case	Defining Condition	Desired Trend	Goal
	$\lvert\text{EIS}\rvert = \lvert\text{System}\rvert$	Condition should never occur	$\lvert\text{EIS}\rvert = \lvert\text{System}\rvert$ in 5 % of impact analyses
	$\lvert\text{EIS}\rvert < \lvert\text{System}\rvert$	Condition should always be met	$\lvert\text{EIS}\rvert < \lvert\text{System}\rvert$ in 70 % of impact analyses
	$\lvert\text{EIS}\rvert \ll \lvert\text{System}\rvert$	Condition should always be met	$\lvert\text{EIS}\rvert \ll \lvert\text{System}\rvert$ in 25 % of impact analyses

Table 3.1 – Relationships between the EIS and whole system [AB93]

Arnold and Bohner studied the relationship between the EIS and the complete software system, examining a similar relationship to Jaccard distance. A sufficiently sharp IA approach should yield results similar to the desired trends and goals defined in Table 3.1 [AB93]. Conversely, given a good IA approach, Table 3.1 can be used to classify changes to software according to their magnitude.

3.3.2 Changed Lines of Code

Lines of code, or source lines of code (SLOC) to be more precise, are a simple but popular metric to measure the size and complexity of software projects. Since lines of code are the direct output of programming, they are often used to measure the size or impact of a change [Ngu+07]. Even though changed lines of code is not a measurement derived from the GRG or diffgraph, it is still included in this thesis as a baseline metric.

Over time, many suggestions were made on how to correctly count SLOC. The main issue is how to distinguish physical and logical lines of code: Physical lines include everything, including blank lines or comments that do not contribute to the program. Logical lines represent a filtered view that normalizes lines with little significance (blank lines, parentheses, etc.) and high significance (many statements in one line) [Ngu+07].

Changes between two source files are usually determined with tools like `diff`.[19] However, this tool has some shortcomings: Changes are only indicated by added and/or deleted lines. There are approaches to combine pairs of deleted and added lines to changed lines [CCDP07].

In this thesis, the changed lines of code metric is implemented as a simple basis of comparison to the size of the SIS. This metric is only used to roughly classify the size of the changes. As already mentioned, the amount of added and deleted lines will be extracted using `diff`. Since this tool does not explicitly mark changed lines, the sum of added and deleted lines will be used as an approximation of changed lines of code. When comparing two versions of a program, `diff` also shows the changes to non-source files such as documentation. This measurement will be called *changed lines* from now on.

Distinguishing changed lines from changed SLOC is a research field of its own and not in the scope of this thesis. When comparing impact metrics to changed lines of code it is still desirable to distinguish changed SLOC from changed lines. Therefore, the tool `cloc` will be used to distinguish changed physical lines from changed lines of code (note: not logical lines of code!). The tool is set up to only count changed, added and removed lines in C source and header files, ignoring empty lines. Even though this is not the accurate name when considering other publications, this measurement will be called *changed SLOC* in the following, since at least it only counts changed lines in source files.

3.4 Node-Specific Metrics

By using the diffgraph as a richer representation of the SIS and EIS, interesting statements can be derived about the affected nodes of a change. The possible research questions can be varied: How much does a single node (function or variable) affect the entire program? How strongly does a node depend on the rest of the program? How much does a single node change internally for a given change to the entire program? The following section provides various metrics to answer these questions.

[19]Unix diff: `https://pubs.opengroup.org/onlinepubs/9699919799/utilities/diff.html`

3.4.1 Position-Dependent Measurements

In a program, not all functions and objects are to be considered as equally important. Central data structures and functions are used in many places of a program. A change to these elements can cause unwanted side effects that affect the entire program. Other functions, such as `main`, are very rarely called or used. Nevertheless, they also play an important role for the entire program, since they often control central processes.

It can be helpful to identify such influential nodes in changes to examine them more closely for unintended side effects. The influential nodes will be detected by examining the diffgraph of a change. In this thesis, a node is classified as influential according to the following criteria:

- A node has many incoming edges: If a function or variable is called or accessed across many functions, a change to this node could indirectly cause a changed behaviour in many of those who access it.
- A node has incoming edges from nodes that are located in different translation units: If a node has many incoming edges inside the same translation unit, its impact is likely only concentrated on this translation unit. A node that is however accessed from many object files is likely to be an important part of the overall program structure.
- A node is located close to the diffgraphs root: Functions and variables that are located close to the `main` function are likely to be concerned with a programs core functionality or structure.

The validity of these assumptions will be examined in Chapter 4.

The presented metrics can be easily measured in the diffgraph. Since those measurements have little meaning in isolation, other nodes have to be examined for comparison. Two approaches to generating a scale for classifying the measured values are possible: 1) For each node in a global reference graph, measure the aforementioned impact values. 2) For each node in the starting impact sets across multiple changes, measure the impact values.

In this thesis, the second approach will be the preferred one. The goal of this metric is to get a sense of the relative impact a changed (or added/deleted) node has on the entire program. Therefore, the impact values will be compared across nodes appearing in starting impact sets.

The metric "number of incoming edges" may not be well suited for each node in a global reference graph or diffgraph, since highly impactful functions may be encapsulated inside of other functions. The number of incoming edges to such impactful functions may therefore be considerably lower than the number of incoming edges of their encapsulating functions. Therefore, in addition to the already mentioned metrics, a transitive closure will be calculated for each GRG. Using this transitive closure, the number of directly and indirectly impacted nodes for each changed node can be determined. This approach is very similar to generating an impact set using static call graphs. However, the number of impacted nodes is calculated individually for each element of the SIS and not combined into an EIS for all changed nodes.

The measurements proposed in this subsection can be used to answer the question on how strongly a change to an individual node could impact the whole programs behaviour.

3.4.2 Similarity Measurements

The metrics proposed up to this point describe how severely a change to a source code definition could impact a programs behaviour. However, besides the size of the SIS, not much has been said about how much of a program has actually changed.

An ideal solution would represent the semantic structure of a node in its local (and global) hash. The magnitude of a change could then be derived from the numerical distance of the hashes before and after the change. Currently, the chosen approach for generating impact sets, *c*Hash, uses a hashing algorithm which has high sensitivity and entropy to small changes. For several reasons, it would not be practical to replace the current hashing algorithm: 1) The *c*Hash and TASTING approaches rely on the fact, that hash collisions are nearly impossible to occur, otherwise the approaches would not be able to output correct impact sets. With a hashing algorithm that is less sensitive to change, collisions can no longer be ruled out. 2) A hashing algorithm that represents the internal structure of a node, which is itself a tree (AST), would be hard to find. The multi-dimensional nature of a tree would have to be mapped into a single-dimensional value which has to be similar to another hash value, if the underlying trees are similar.

In order to still be able to make a statement about the internal change of a node, the following methods are presented.

Successor Similarity

Since the hash of a node does not convey information about its content, this information must be obtained in some other way. In fact, the GRG and diffgraph already contain part of a nodes internal structure: The outgoing edges of a node (in this case a function) can be used to determine which other definitions are used by that node. When the set of child nodes changes, a degree of internal change can be derived.

To measure this internal change, the Jaccard index is used again. This metric is applied to each node that has been modified in a change, i.e. the SIS. For each node in the SIS, a set of child nodes is obtained from the old and new GRG. These sets are then used to determine the Jaccard index and/or distance between a nodes dependencies. In the case of added or deleted nodes, the missing set will be replaced by an empty set.

The resulting metric is called *successor similarity*. An index of one means that a nodes dependencies have stayed the same. A lower index indicates that a certain amount of dependencies has changed. However, this metric cannot quantify a functions change in behaviour.

Sub-AST Tree Edit Distance

To get a more detailed insight into the internal structure of a node, additional information is required. In principle, a node in the GRG or diffgraph is a proxy for a subordinate AST. The structure of an AST changes only if the corresponding program code changes semantically. This property is already exploited by *c*Hash when generating the local hashes. In order to be able to quantify the internal change of a node, the corresponding Sub-AST is compared before and after a change. For this comparison, the tree edit distance algorithm, which was already described in Section 2.2.2, is used. Note, that the TED algorithm is not applied to the GRG in this case, but only to the ASTs which are represented as nodes in the GRG.

For each node in the SIS, the underlying AST is stored before and after applying a change. The ASTs are extracted from the source code with the clang C compiler.[20] The edit distance between those Sub-ASTs resulting from the TED algorithm quantifies the influence of the change at the examined nodes as an absolute value. Changes to variable names and their values, to instructions, operations and their sequence and all other semantically relevant edits are tracked as either additions, changes or removals of nodes in the AST. These measurements of change are called *Sub-AST Tree Edit Distance* (Sub-AST TED).

The Sub-AST TED is a node-specific metric. However, by accumulating the edit distances for all nodes in the SIS, Sub-AST TED can also be used as a program wide change metric.

[20]Dump the AST for a C source file as JSON: `clang -Xclang -ast-dump=json -c <source>.c <includes>`

4 EVALUATION

In the following chapter, the proposed impact analysis metrics are examined in more detail. First, the program wide and position-dependent metrics are checked for their expressiveness, i.e., whether they are sufficiently sensitive and sharp. Subsequently, the Jaccard distance is used to examine whether similar patterns can be found in the development history of the open source projects QEMU, CPython, OpenSSL and Lua. Furthermore, it is examined whether there is a correlation between the impact of a change and the social interaction around this change. Last, the node-specific similarity measures are examined and it is investigated whether they have any relationship with conventional change measures.

To better assess the statistical relationship between different measurements, various correlation coefficients are used in this thesis. Pearson's r is used to estimate the linear relationship between two variables. This coefficient is only useful for normally distributed datasets and will be applied if a linear correlation is suspected in the datasets. For non-normally distributed data (with significant outliers), Spearman's r_S and Kendall's τ can be used. These do not measure whether the variables are linearly related, but whether they can be described well by a monotonic function. For datasets with repeating data points, Kendall's τ provides a tighter estimate. Table 4.1 shows the keywords used for interpreting different correlation coefficients [Ako18].

Correlation Coefficient		Interpretation
1	-1	Perfect
0.9	-0.9	Very Strong
0.8	-0.8	Very Strong
0.7	-0.7	Moderate
0.6	-0.6	Moderate
0.5	-0.5	Fair
0.4	-0.4	Fair
0.3	-0.3	Fair
0.2	-0.2	Poor
0.1	-0.1	Poor
0	0	None

Table 4.1 – Interpretation of different correlation coefficients [Ako18]

4.1 Generating Impact Metrics

This section describes the test setup and implementation used to investigate the proposed methods. All of the projects studied in this chapter are managed with the SCM tool `git`, in which one commit corresponds to one change. First, a description is given of how a set of changes is extracted from these code repositories. Then, the generation of impact sets for each of these changes is described. In the next step, the impact sets are used to quantify the severity of the changes. The metrics can then be visualized and explored graphically.

Figure 3.1 gives an overview over the described change impact analysis workflow. The different steps of impact analysis have been automated in a tool called `impact-analyzer`. The following subsections explain the implementation of those steps.

Selecting Changes for Impact Analysis

A significant amount of modern open source projects use Git as their SCM tool. Since Git offers a simple and uniform interface for accessing the development history, this thesis only studies projects that are managed with this tool. Nevertheless, the approaches shown can also be applied to other projects with similar SCM tools.

The development history of a project is represented in Git as a directed acyclic graph. Each node represents an independent version of the software (*commit*), each outgoing edge a change to this version, which results in a new commit. The development history can *branch* out to make independent changes to different parts of the software. These branches can later be *merged* again, the resulting commits with multiple incoming edges are called merge commits.

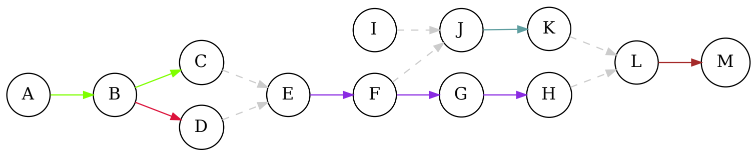


Figure 4.1 – An example of a Git development history

For the impact analysis in this thesis, only linear changes are examined. Merge commits are excluded because they include multiple original program versions. The presented IA approach can only determine the difference between two program versions, which is not compatible with the semantics of merge commits. Figure 4.1 shows an example of a Git development history. All incoming edges to merge commits are marked grey, those changes will not be considered for impact analysis. The colored edges are part of linear commit series, where for each commit the GRG will be constructed for further inspection. Note commit node `I`: For this commit, no GRG will be constructed, since it is not part of a linear commit series. If a commit is known to not be compilable, it can also be excluded from impact analysis.

The linear commit series relevant for impact analysis are stored in order, which is relevant when generating the impact sets. The commit series can be automatically extracted from a Git repository using the `commits` subcommand of `impact-analyzer`.

Generating Impact Sets

The process of building the global reference graphs and diffgraphs for a project can be automated with the `impact-analyzer` subcommand `gen` (see Figure 3.1). At this point it is important that the commit series are stored in order. For each commit, the program and the GRG are built as described in Section 3.2.1. If this is done in an orderly manner, the underlying build system can reuse more of the already compiled object files, so the compilation processes takes less time.

Not each software project employs the same build system. For this reason, an interface was created for configuring each project to be examined individually. For each project, the compiler and linker arguments required for *c*Hash and `chash-global` are configured and passed to the build system. In addition, the commits that cannot be compiled are also stored in the configuration and skipped when building the GRGs and diffgraphs.

Calculating Impact Metrics

After generating the GRGs and diffgraphs, all program wide and node specific metrics for each change can be computed with the `impact-analyzer` subcommand `stat` (see Figure 3.1).

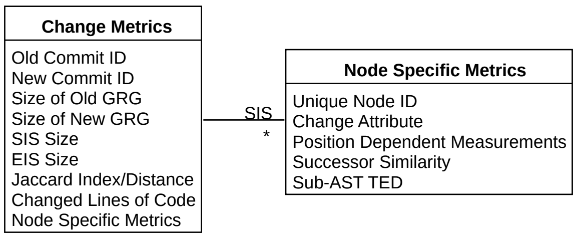


Figure 4.2 – A summary of all metrics for quantifying software change

Figure 4.2 shows a summary of all metrics that are used to quantify the change to a program between two commits. In order to distinguish the measurements for each change, the commit IDs of the original and resulting commit are stored. For each node in the SIS, its unique identifier is stored together with the node specific metrics. All computations carried out on graphs, as described in Sections 3.3 and 3.4, were implemented with the Python NetworkX package.

The measurements on changed lines of code are made using the `cloc` tool. The tool can be configured to compare two Git revisions, i.e., commits, and output the changed, added and removed lines as a report in machine-readable formats such as JSON.[21]

The TED algorithm chosen for the Sub-AST tree edit distance metric is AP-TED, with an implementation by Pimentel [Pim17; PA16]. The cubic runtime encountered in the worst case for TED is acceptable, since the AST of a function rarely becomes unmanageably large. However, it is possible to set an upper limit for the size of the Sub-ASTs. The Sub-AST TED metric is then aborted in individual cases with a warning.

Exploring the Metrics

The measurements from the previous steps can be evaluated graphically with the `plot` subcommand of `impact-analyzer`. It is also possible to compare the measurements with datasets generated by other studies on open source projects. The results of change impact quantification are discussed in this chapter using graphs generated with `plot`.

[21]`cloc --git --diff <rev1> <rev2> --include-lang=C,"C/C++ Header" --json --quiet`

4.2 Target Projects

The metrics are applied to several open source projects of different scales (QEMU, CPython, OpenSSL, Lua; see Table 4.2) so that the conclusions are not distorted by the peculiarities of a single project. As already mentioned in Section 2.4, the studies can only be performed on software written in the C programming language. For each project, a range of recent changes has been selected. Since in some projects the build system changes over time, commit ranges were chosen for the analyses in which the build system remains stable. Accounting for changing build systems was not considered necessary because the datasets examined contain enough changes.

Project	Configuration Target	Nodes in GRG	Source Files	Examined Changes
Lua		979	35	1,817
OpenSSL		9,791	722	1,927
CPython		11,281	229	1,295
QEMU	x86-64	13,897	621	12,373

Table 4.2 – Overview over the examined projects and their average size over all commits

Table 4.2 shows all projects that were analyzed in this thesis. The number of nodes in the respective GRGs ((static) functions and global variables) and the number of source files are an average value over all examined commits. The reported amount of source files includes all source files that contribute to the GRG. This amount is extracted from the unique identifiers in a GRG. The examined commit ranges for each project can be seen in Appendix A.1. The projects were chosen for multiple reasons: They represent a wide spectrum of software classes, i.e., hypervisor (QEMU), interpreter (CPython and Lua) and library (OpenSSL). Lua in particular is a small and quickly compiling project, which is ideal for rapid prototyping of new metrics. For QEMU and CPython, additional datasets on social interaction and effort estimation were available, which will be examined in Sections 4.4 and 4.5. This also explains the relatively large amount of changes examined for QEMU: The dataset chosen to measure social interaction contains a similarly large number of data points that should not be discarded.

The build systems of many projects can be configured to customize the resulting program. This can involve debug or release configurations, additional features, or different target architectures. Unless stated otherwise, all projects examined were compiled with their default configuration and as a release build. The QEMU project was compiled for the x86-64 architecture only. As a result, it must be taken into account that the presented IA approach is less sharp for this project: Changes to architecture-specific code that do not affect x86-64 are not reflected in the GRG and therefore ignored during impact analysis. The effects of only building for one architecture can be seen in Table 4.3. For QEMU, less than half of the C source files are compiled into object files.

Figure 4.3 gives a visual overview of the average GRG sizes of the projects studied relative to the average and median SIS (changed local hashes) and EIS (changed global hashes) across all changes. The set sizes are scaled logarithmically, so the smaller sets stay visible. Several observations are noteworthy: 1) The size of the authored changes (SIS) is similar across all projects. Only OpenSSL tends to have smaller change increments. 2) Only for QEMU is the average size of the EIS not in the same order of magnitude as the size of the overall system. 3) For QEMU, the impact sets lay one order of magnitude apart, for Lua it is two orders and for CPython and OpenSSL the impact sets lay three orders of magnitude apart. The deviation between average and median IS sizes result from their irregular distribution, which is inspected in detail in the following sections.

Project	.c files	.c files after build	.o files after build	$\frac{.o}{.c}$ after build
Lua	35	35	34	97.14 %
OpenSSL	813	813	760	93.48 %
CPython	326	327	239	73.09 %
QEMU	2,307	2,556	1,139	44.56 %

Table 4.3 – Ratio of object and source files in the examined projects. The files were counted with "`find . -type f -not -path "./<test>/*" -name "*.<c/o>" | wc -l`" for the most recent of all examined commits. In some projects, additional source files are generated during the build.

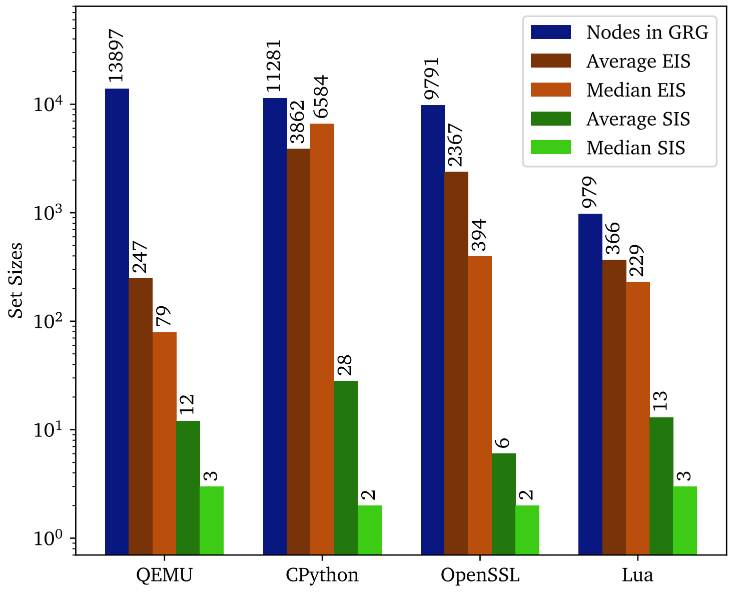


Figure 4.3 – Mean and median impact set sizes compared to the systems size of four projects. Changes with no impact ($|SIS| = 0$) are excluded. No impact is measured if the GRG is not changed, e.g. in case of changes to the documentation or code that is not relevant to the GRG.

These differences in the sizes of the sets are particularly interesting with regard to Table 3.1. Only in one out of four projects (QEMU) does the presented IA approach regularly deliver impact sets that are significantly smaller than the overall system. Furthermore, only in QEMU are the EISs only slightly larger than the SISs. In the following, it is discussed whether these differences result from the fact that the selected IA approach is unsuitable according to Arnold and Bohner, or whether the size of the EISs in Lua, OpenSSL and CPython result from strong ripple effects within those projects.

The goal of this thesis is not only to quantify the impact of a change correctly, but to do so efficiently. In order to better understand the time required for impact analysis, a projects clean build was compared with the average build times for all the commits examined. All tests were performed on the same machine (96-Core Intel Xeon Gold 6252 @ 2.10 GHz). Figure 4.4 shows the build times of the four projects. First of all, it is noticeable that for QEMU and CPython, the clean build takes longer than the complete impact analysis. This is due to the fact that during the first build (but after the configuration scripts have been run), additional dependencies have to be created, which cannot

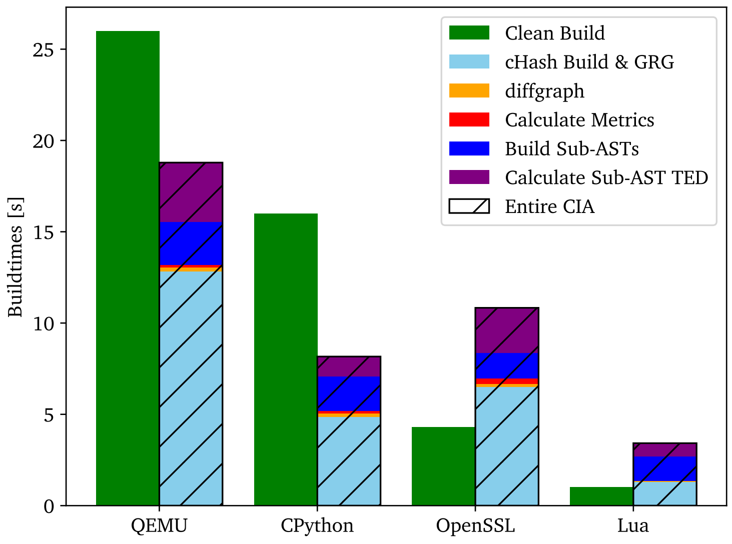


Figure 4.4 – Build and impact analysis times for four open source projects

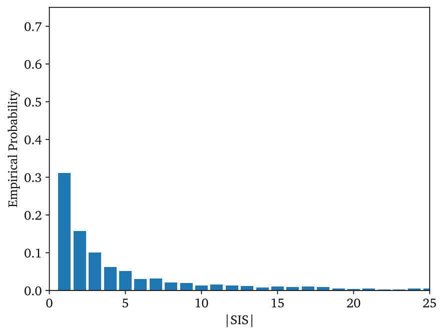
be created in parallel. Therefore, in those cases, the clean build performs worse than the build with *c*Hash. Generating the GRG and other data required for IA creates a measurable overhead with OpenSSL and Lua. Furthermore, it is noteworthy that the time required to generate and analyze the Sub-ASTs is similar for all projects. This is because the amount of Sub-ASTs examined does not depend on the size of the GRG, but on the SIS, which is similar across all projects. In practice, build times with *c*Hash could be lowered if it is directly integrated with the build systems. Currently, the build times also include the time spent on configuring the projects to use *c*Hash and `chash-global`. However, the additional overhead for impact analysis is deemed acceptable.

4.3 Assessment of the Impact Quantification Metrics

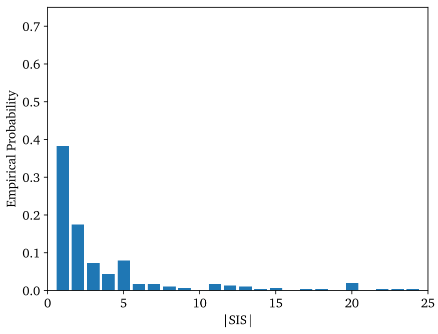
Before comparing the impact analysis metrics to social interaction and effort estimates, the significance of the impact quantification metrics is examined. The measured values are then used to explain the relationships shown in Figure 4.3. In the following, only changes with $|\text{SIS}| > 0$ are considered. This excludes changes which are only concerned with formatting or documentation updates.

First, it is examined whether the magnitude of the authored changes is subject to a natural distribution. Figure 4.5 shows the empirical probability distribution for SIS sizes across multiple changes and projects. The distribution can be described reasonably well with the Pareto probability density function, as in Equation (4.1) [Sch12]. Other natural datasets, such as the distribution of words in spoken languages or the number of inhabitants in cities, also follow similar laws [Tao09]. Thus, it can be assumed that most of the changes are man-made and that the measurements are not distorted by automatically generated changes. The SIS sizes are not distributed normally, therefore Pearson's r is not suitable for comparing them with other measurements.

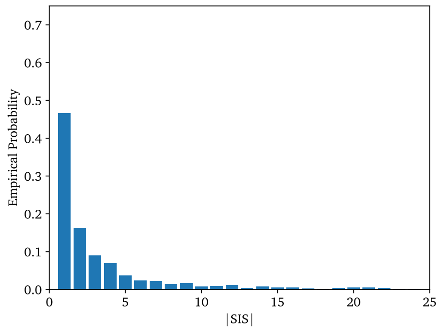
$$f(x) = \begin{cases} \frac{\alpha x_{\min}^{\alpha}}{x^{\alpha+1}} \overset{x_{\min}=1}{=} \frac{\alpha}{x^{\alpha+1}} & x \geq x_{\min} \\ 0 & x < x_{\min} \end{cases} \tag{4.1}$$



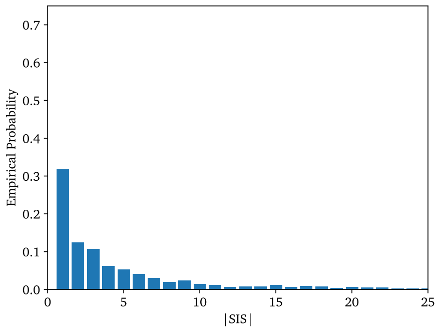
(a) QEMU. $n_{|\text{SIS}|>0} = 2{,}076$; $\alpha = 0.33$



(b) CPython. $n_{|\text{SIS}|>0} = 304$; $\alpha = 0.39$



(c) OpenSSL. $n_{|\text{SIS}|>0} = 868$; $\alpha = 0.46$



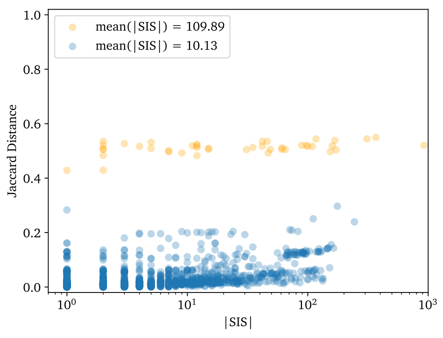
(d) Lua. $n_{|\text{SIS}|>0} = 1{,}385$; $\alpha = 0.32$

Figure 4.5 – The empirical probability for the size of the starting impact set across multiple changes for different software projects. The plots are truncated at $|\text{SIS}| = 25$, with some outliers beyond that. Changes with no impact ($|\text{SIS}| = 0$) are excluded.

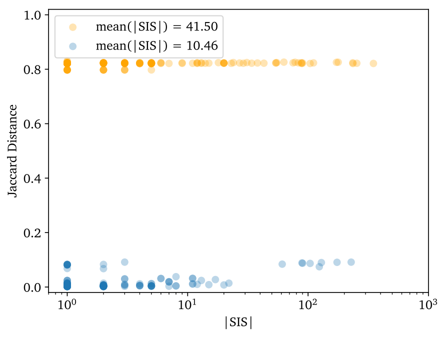
4.3.1 Relationship Between the Metrics

Before the impact quantification metrics are compared with other change measurements or used in practice, it is verified that the metrics are not in a linear relationship with each other. If two measurements are highly correlated, the usefulness of one of the two metrics may be questioned, as both would interpret a change in the same way. The relationship between the SIS and EIS is considered first, as these are the most common measurements in impact analysis. Figure 4.6 shows the relationship between the size of starting impact sets (|SIS|) and the corresponding Jaccard distance for each change. The variables are clearly not linearly dependent. Changes that initially affect only a few nodes in the GRG can therefore also potentially have a major impact on the entire program. However, the measured coefficients indicate that they correlate fairly when described by some monotonic function. This can be explained by the fact that a larger SIS naturally affects more nodes in the GRG transitively, resulting in a larger EIS.

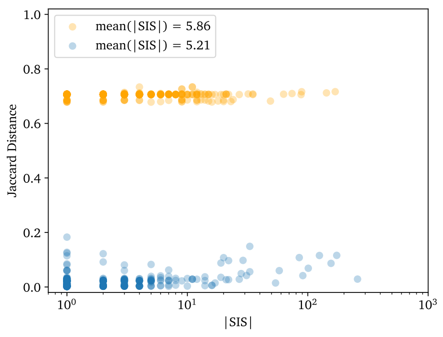
Nevertheless, it would be difficult to find a monotonic function that explains the strong clustering of low- and high-impact changes. In all projects, it can be observed that there are changes that affect the entire software only up to a certain threshold. With some distance, other changes accumulate that result in a much larger EIS. The measured Jaccard distance in both clusters is only weakly dependent on |SIS|, as shown by the average |SIS| for the clusters from the graphs. For OpenSSL,



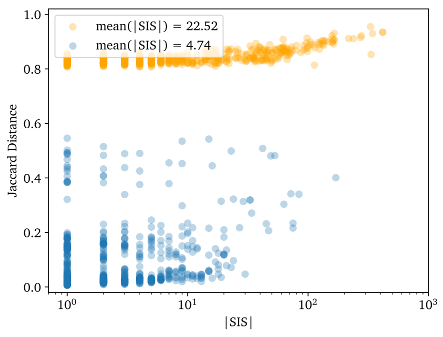
(a) QEMU. $r_S = 0.51$; $\tau = 0.38$



(b) CPython. $r_S = 0.25$; $\tau = 0.19$



(c) OpenSSL. $r_S = 0.34$; $\tau = 0.25$



(d) Lua. $r_S = 0.48$; $\tau = 0.36$

Figure 4.6 – The relationship between the size of starting impact sets and the Jaccard distance (basically the normalized size of the EIS, see Section 3.3.1) for different projects. The relationship is visibly nonlinear, so only Spearman's r_S and Kendall's τ are presented. |SIS| is scaled logarithmically so the dataset can be observed well across all orders of magnitude. Low- and high-impact changes are colored differently.

the average |SIS| for the high impact cluster is even lower than in the low impact one. The strong separation between low- and high-impact changes can have several causes. In the following, high-impact changes and possible reasons for their classification are examined: 1) Changes to record data types. These are not part of the GRG, but affect nodes that depend on them, such as functions that access structs. 2) Changes to macros. These are also not part of the GRG, since they are substituted by the preprocessor before syntax analysis, but may affect a large portion of the program. 3) Changes to high-impact nodes. Those are functions or variables that have a high score on any of the node-specific position-dependent measurements.

The reasons for a change being classified as high impact are studied more closely with Lua. The results are also applicable to the other projects, which are not shown here in order to keep the chapter shorter. It is first checked whether changes to structs lead to a change being rated with a high Jaccard distance. Figure 4.7 shows the relationship between |SIS| and Jaccard distance for Lua, as seen in Figure 4.6d, but grouped by changes that do and do not modify a record data type. The number of high impact changes relative to all changes is slightly larger for changes which modify a record data type. Nevertheless, changes that do not affect record data types still result in a large number of high impact commits. It can thus be assumed that changes to record data types are not

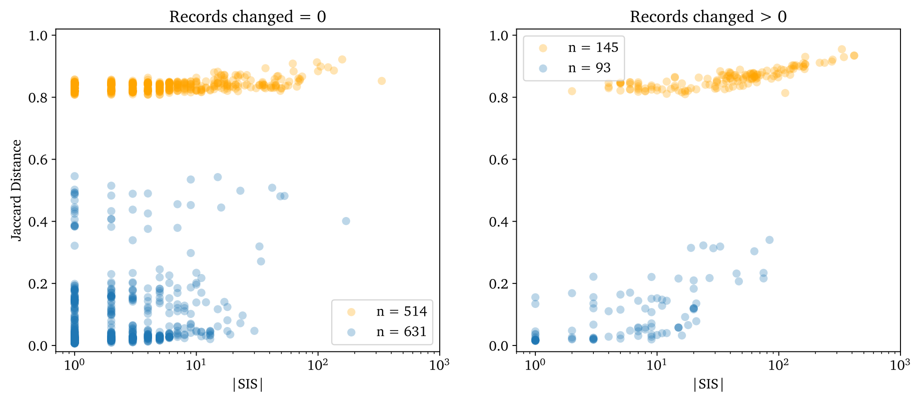
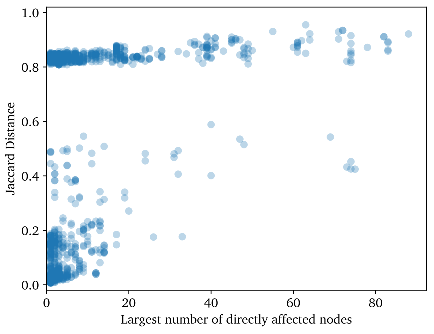


Figure 4.7 – The relationship between |SIS| and Jaccard distance for Lua, grouped by changes that do or do not modify record data types. The number of high impact changes relative to all changes is slightly larger for changes which modify a record data type: $\frac{n_{\text{records changed=0; high impact}}}{n_{\text{records changed=0}}} = 0.45$; $\frac{n_{\text{records changed>0; high impact}}}{n_{\text{records changed>0}}} = 0.61$

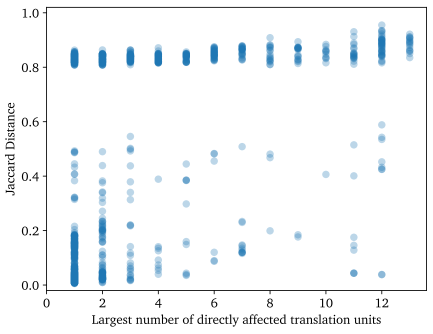
the (sole) reason for high impact changes. The same is true for changes to macros: Both macros and record data types are not captured in the GRG. A change to these will possibly result in a large SIS, as described in Section 3.2.2. Since SIS and EIS are not linearly correlated, changes to structs or macros are not the cause for a large EIS (or high Jaccard distance).

It is now checked whether individual high impact nodes (functions/variables) in the GRG are responsible for particularly large EIS. Figure 4.8 shows the dependency of the Jaccard distance for a change on the different position-dependent metrics presented in Section 3.4.1. For each change only the most impactful node, according to the metric, is displayed. The other nodes are ignored, as the purpose is to check whether a single node potentially overshadows the impact of all other nodes. The plots show a clear trend: The size of the EIS is mostly independent from the measurements that can be directly taken from the diffgraph. This can be explained by the fact that the measurements only describe the local dependencies of a node in the diffgraph. The distance of a node from the root also does not seem to be decisive. A deep-lying node does not necessarily influence many other nodes: It can be a chain of function calls that is not branched any further. The best (near perfect) correlation can be found in the largest size of a changed nodes transitive closure. This was to be expected, but still has some interesting implications: A single node can indeed overshadow the influence of all other nodes in a change. It remains to be tested whether a combination of the position-dependent metrics (without transitive closure) can explain the influence of individual nodes on the GRG. However, since no particularly good correlation was found for any of the metrics, this was not pursued further.

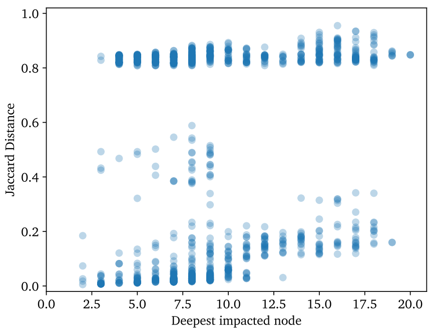
Although the transitive closure is able to explain the size of the EIS well, it also has a serious disadvantage, which only became apparent when testing with projects other than Lua: The algorithm for calculating the transitive closure has a runtime complexity of $O(n^3)$ with the Floyd-Warshall algorithm, and $O(n^{2.376})$ with the best algorithm employing matrix multiplication [BJG09]. While the computation for Lua (979 nodes) runs in only 3.75 s on the test machine, the computation for CPython (11,281 nodes) already amounts to 11.86 m. Thus this metric, in contrast to the GED, is practically computable, but far from this thesis' goal of an efficient impact analysis. Nevertheless, the data obtained was used to get an overview of the most influential source files containing the most influential GRG nodes of the different projects, which can be seen in Table 4.4 for CPython.



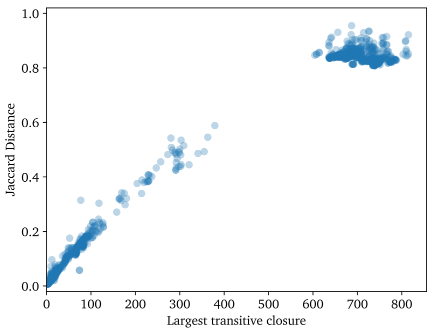
(a) $r = 0.41$; $r_S = 0.67$; $\tau = 0.51$; Fair correlation



(b) $r = 0.46$; $r_S = 0.6$; $\tau = 0.47$; Fair correlation



(c) $r = 0.19$; $r_S = 0.28$; $\tau = 0.22$; Poor correlation



(d) $r = 0.99$; $r_S = 0.84$; $\tau = 0.69$; Very strong cor.

Figure 4.8 – The relationship between the Jaccard distance (|EIS|) and different position-dependent measurements (see Section 3.4.1) for the most impactful node in a change. The data is pulled from the Lua project.

Those high impact nodes are all nodes with a large enough transitive closure to be contained in the high impact cluster, as in Figure 4.8d.

Filename	High impact nodes	Source file description
`Python/ast_unparse.c`	36	AST unparser
`Python/symtable.c`	30	Symbol Table handling
`Python/pylifecycle.c`	18	Python interpreter top-level routines
`Python/initconfig.c`	16	Python initialization
`Python/hamt.c`	11	Implementation of an immutable mapping
...	...	
33 of 229 files contain high impact nodes	166 of 11,281 nodes have high impact	

Table 4.4 – Overview of the most influential source files for CPython. The files are sorted by the number of high-impact nodes (large transitive closure) they contain.

Table 4.4 shows that the most influential nodes of a project's GRG are bundled in just a few files. Moreover, only a small portion of the nodes are even classified as high impact nodes. In the example shown, all source files contribute to the core functionality of the program. Thus, it can be assumed that nodes that trigger a large EIS are actually crucial to the functionality of a program. Generally, these observations also apply to the other examined projects.

4.3.2 Comparing Open Source Projects

The presented impact metrics can be used to study the structure and development behaviour of different projects. In addition to a statement about the internal structure of these programs, it can also be checked whether the IA approach presented is sufficiently sharp, i.e., whether the EIS is not constantly being overestimated too aggressively. Figure 4.3 already gave an overview of the scales of change in different projects. However, it was not clear from the mean values of the change sizes how the changes are actually distributed. The figures in Section 4.3.1 showed that changes of similar scales occur in clusters, but not how frequently such changes actually occur. The empirical distribution of Jaccard distances across multiple changes in different projects is shown in Figure 4.9. It is interesting to observe that in Lua, OpenSSL and CPython about half of the changes have a small to medium EIS and the other half cause a large EIS. In QEMU, this distribution is much more gradual and only a small portion of the changes are high impact, with the largest EIS still significantly smaller than in the other projects.

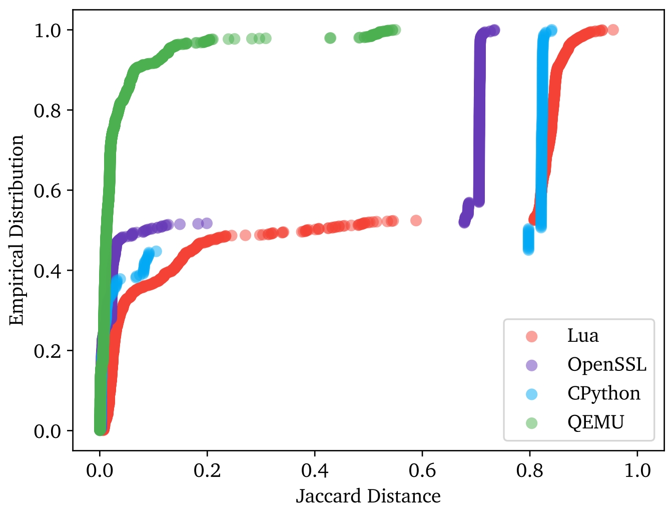


Figure 4.9 – The empirical distribution of change impact measured with Jaccard distance for different open source projects.

If QEMU is ignored in the evaluation, the chosen impact analysis approach could be accused, according to Arnold and Bohner, of generating unnecessarily large EISs and thus of being unsharp [AB93]. However, QEMU is a good counterexample to defend the approach, since for this project the distribution of the EIS sizes satisfies the requirements of Table 3.1. Low internal coupling is a desired quality for software systems since it improves understandability, testability, maintainability and reliability [OHK93]. The size of EISs is a good measure for this property, since it measures the dependencies between the SIS and all other definitions of a program. This allows the statement to be inverted: The chosen impact analysis approach is sharp and results in high Jaccard distances because the other three projects are not as robust to change as QEMU. Even small changes often have a potentially large impact on those programs.

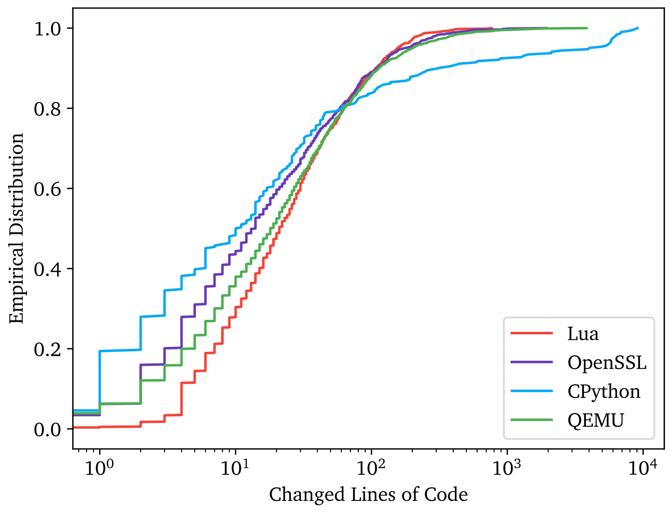
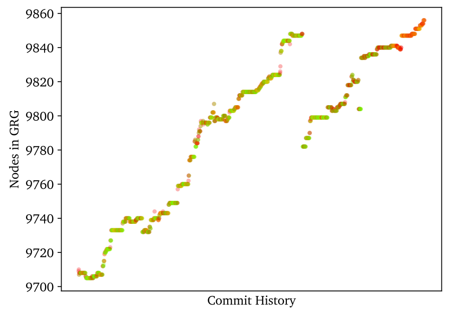


Figure 4.10 – The empirical distribution of changed SLOC (see Section 3.3.2) for changes with $|\text{SIS}| > 0$ across different open source projects.

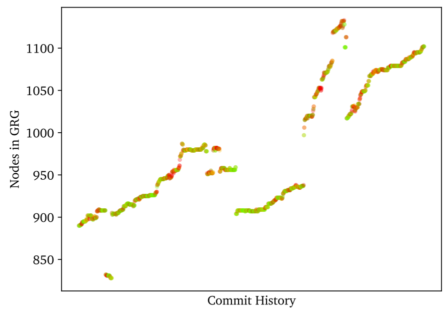
In order to rule out the possibility that the differences in the sizes of the EISs result from changes of different sizes to the projects, the changed SLOC were compared as a basic metric for each project. Figure 4.10 shows that the number of changed lines of code tends to be similar across all projects. Only CPython is an outlier here: The amount of large changes results from the fact that some source files (such as `parser.c`) are generated automatically. As a result, source files can change significantly even after minor adjustments. It is also notable that there are changes that seem to occur without changing lines of code. This is because the employed SLOC metric only counts changes to C source and header files. In some projects, other files also contribute to the GRG, e.g. by creating intermediate source files in the build process.

The time spans examined for the various projects extend over several years. Therefore, the impact analysis datasets are also suitable for conducting long-term analyses of the projects. With the data obtained during IA, an attempt was made to reproduce Wu's studies on punctuated software evolution. According to this theory, software evolves in long periods of small changes to the architecture and short periods of large changes. Wu examined three different open source projects: OpenSSH, PostgreSQL and Linux. The studies were carried out using the *evolution spectrograph*: The size of a project (measured by the number of files) was plotted over discrete time steps (commits to the SCM). The intensity of a change was measured by comparing the Fan In/Out of dependencies of the files. Wu concluded that punctuated software evolution can indeed be observed in the projects studied. He noted, however, that the results may be biased by the coarse granularity of his studies [Wu06].

The presented impact analysis approach can be used to repeat those studies with much finer granularity. The time steps are still measured using commits. The size of a project is measured by the number of nodes in a GRG. The intensity of a change is measured with the size of a commits SIS. Figure 4.11 shows the new interpretation of the evolution spectrograph. It is possible to observe



(a) OpenSSL



(b) Lua

Figure 4.11 – Evolution Spectrographs to discover Punctuated Software Evolution in different open source projects. The size of a project is traced over time. The intensity of changes is visualized using different colors: Small SIS $\rightarrow$ green; Mid sized SIS $\rightarrow$ yellow; Large SIS $\rightarrow$ red.

longer periods of green and yellow (small) and short bursts of red (large) changes. Upon closer inspection, large changes do not necessarily lead to an increase in the projects size. Increases in project size are however always caused by commits with a large SIS. The separation between the high and low impact periods is not as pronounced as in Wu's doctoral thesis. There may be several reasons for this: 1) The chosen IA approach is more fine-grained than the pure consideration of file dependencies and thus does not react as strongly to newly added or deleted files. 2) The time periods observed in this thesis contain less commits than the ones observed by Wu. This means that longer development cycles cannot be observed as well.

Figure 4.11 only shows the evolution spectrographs for OpenSSL and Lua. Both projects have a very linear development history with only few branches and merge commits. CPython's and QEMU's development history are disrupted by many merge commits, which makes it hard to display their development history in a linear fashion. Therefore, they are excluded from the evolution spectrograph analysis. Note, that the number of nodes in the GRG in both plots is not zero-based. This view was chosen to show the development histories in more detail.

This brief venture into the study of long-term trends within projects was made to demonstrate that the results of IA can also be used for such studies. However, the additional precision of these analyses comes at the price of having to perform IA for each change, which can take several hours or even days for longer development histories.

4.4 Correlating Mailing List Discussions and Impact Metrics

The significance of the individual impact metrics is difficult to validate in isolation. The evaluations to this point have attempted to contextualize what the individual metrics say about different projects. The metrics can be used to compare the severity of a single change with other changes to the same software. To further investigate the metrics, it is useful to compare them to other datasets that quantify the magnitude of a change. There are some approaches to such comparisons: One is to take other established change metrics as a baseline. This is done in Section 4.5. Another method is to have experts evaluate the severity of a change. Expert opinion is one of the more common methods of assessing the complexity of a system [RMT09]. The experts of a system, in this case the developers, can judge the dependencies within a system and therefore the potential impact of a change well.

Expert opinions can be obtained in numerous ways: A common approach are interviews or surveys. This is not feasible for this thesis, since multiple developers would have to rate thousands of commits for their complexity to gather a statistically significant dataset of rated changes.

Another approach is to use the results of effort estimates as a baseline. At the beginning of a development cycle, developers provide estimates on how complex a task is. A more involved change often has a higher impact on the software, so these estimates can also be seen as a kind of a priori impact quantification. In software engineering, methods such as planning poker are common. Molokken-Ostvold and Haugen show that such approaches can indeed provide accurate estimates of effort [MOH07]. In the context of this thesis, however, no suitable open source C project could be found in which a sufficient number of effort estimates are available.

Nevertheless, in order to obtain an expert opinion on changes investigated with the metrics shown, the following hypothesis is formulated: The intensity of social interactions that arise around a proposed software change is directly correlated with the severity of the change. This is supported by Baysal and Malton, who have shown that discussions and changes are correlated to some degree [BM07]. The assumption is justified as follows: Small, stylistic changes do not require much discussion. In the case of more profound architectural changes or new features, it is expected that they will be discussed at greater length with regard to their correctness and necessity.

Social interaction on open source software can take different forms. Platforms such as GitHub allow issues to be raised about proposed improvements. These can be commented on and reacted to with different emoji. In this thesis, a different data source is chosen: Various open source projects like QEMU or the Linux kernel use mailing lists to discuss changes to the code. The emails on the lists are examined as an indicator of social interaction. For this purpose, data collected by Ramsauer et al.'s patch stack analysis (PaStA) tool is used.[22] PaStA is able to mine software repositories and associated mailing lists and to establish relationships between commits and message threads that otherwise could not be easily found [RLM16; RLM19].

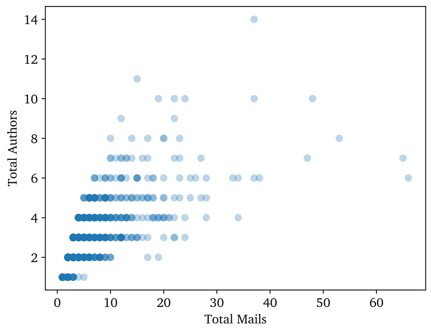
Variable	Explanation
Commit ID	The Git commit hash is necessary to link email data to the impact metrics of a change.
Total Mails	The number of all mails related to the commit from all email threads, excluding automated messages sent by bots.
Total Authors	The number of authors participating in a discussion, excluding bots.
First Message Last Message	The date of the first and last message across all related email threads.

Table 4.5 – PaStA email dataset

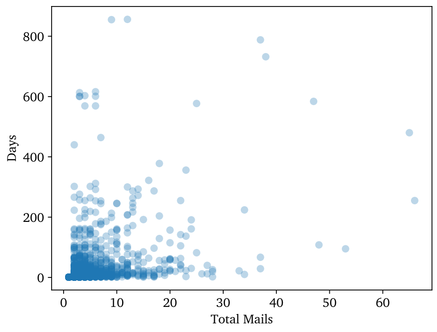
For the proof of concept of correlating mailing list data with impact metrics, the QEMU project was chosen. QEMU is a reasonably sized project and its mailing list has already been mined by the PaStA authors. Table 4.5 shows the variables that were extracted from the email threads to each commit. Each of these values is intended to measure the intensity of social interaction. The number of mails is the most direct measure of the size of a discussion. A closely related measurement is the number of authors. However, this value allows filtering long discussions between two authors. It is weighted more heavily when many people have an opinion on a topic. The length of time between

[22]PaStA on GitHub: `https://github.com/lfd/PaStA`

the first and last message provides another interesting insight: If a change is discussed for a long time between the first submission and the acknowledgement, this may indicate that the change has to go through several iterations and may therefore be more complex.



(a) $r = 0.74$; $r_S = 0.9$; $\tau = 0.81$; Moderate correlation



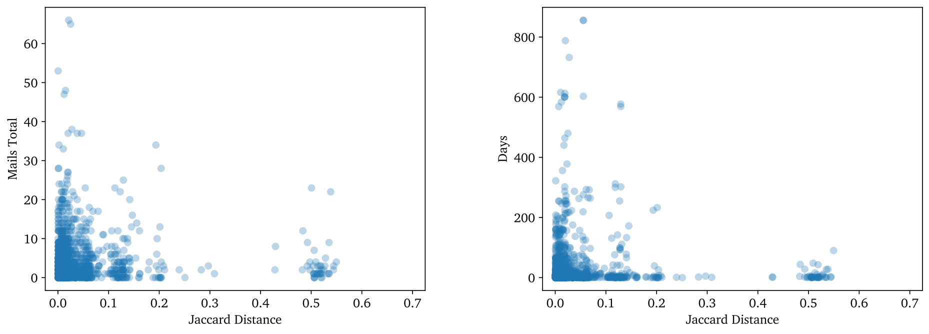
(b) $r = 0.39$; $r_S = 0.73$; $\tau = 0.59$; Fair correlation

Figure 4.12 – Correlation of the email dataset variables with themselves

Before examining the email data for correlation with the impact metrics, it is checked whether the email data strongly correlates with itself. If two of the variables are too similar, one is probably unnecessary. Figure 4.12 shows the comparison of the number of emails to the amount of authors and the duration of the discussions. The number of messages and authors per discussion correlate moderately to very strongly. This is to be expected, since there must be at least as many messages as authors. The number of mails and the duration of the discussion only correlate fairly well. As already mentioned, this may be due to the fact that there may be a varying time of improvement and correction between the first submission of a change and the final accepting message. The impact metrics will be mostly compared to the amount of messages per commit.

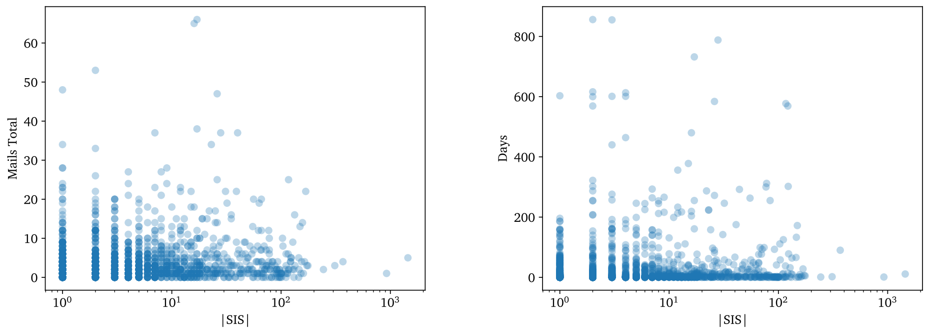
First, the relationship between mailing list discussion and the size of the EIS, represented by the Jaccard distance, is examined. The Jaccard distance best reflects the potential impact of a change on the overall program architecture. According to the hypothesis mentioned at the beginning of this section, developers have the best knowledge of the internal relationships and architecture of a project. As a result, they should be sensitive to and strongly discuss potentially major structural changes. Figure 4.13 shows that |EIS| does not correlate in any way, neither linearly nor in any other monotonic way, with the intensity of a discussion on a change. It could be questioned whether the Jaccard distance is an appropriate measure of the potential impact of a change on the whole project. Especially with regard to high impact changes, Jaccard distance can be described as very sensitive, see Section 4.3.1. However, even if the high impact changes are excluded, the correlation does not improve at all.

The EIS generated by the GRG approach is an overestimate of the actual impact of a change, as described earlier. Thus, the difference from the AIS may be too large to classify a change as significant to the software's architecture. The measurement that actually measures the magnitude of a change with near certainty is the SIS. For this, the former hypothesis has to be rephrased: The more parts of a program are changed simultaneously, the higher the probability of making a mistake. This assumption is supported by Eyolfson et al. who claim that stable commits, which tend to be small, contain fewer bugs [ETL14]. It is therefore expected that smaller changes will be discussed less because they are less prone to errors.



(a) $r = 0.015$; $r_S = 0.028$; $\tau = 0.02$; No correlation **(b)** $r = 0.002$; $r_S = 0.021$; $\tau = 0.015$; No correlation

Figure 4.13 – Relationship between |EIS| and mailing list discussion intensity



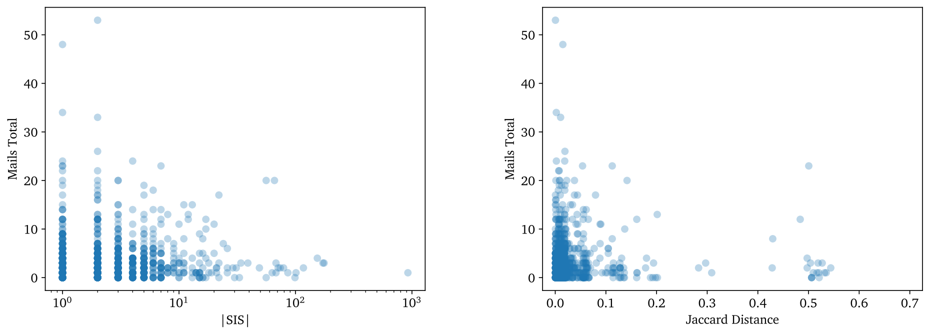
(a) $r = 0.029$; $r_S = 0.11$; $\tau = 0.082$; No correlation **(b)** $r = 0.038$; $r_S = 0.122$; $\tau = 0.091$; No correlation

Figure 4.14 – Relationship between |SIS| and mailing list discussion intensity

However, Figure 4.14 shows that this hypothesis cannot be statistically supported either. The size of the SIS is not correlated with the amount of social interaction in any meaningful way. As described in Section 3.2.2, the SIS may become unexpectedly large if a record data type has been changed. To rule out the possibility that changes to such structures might distort the correlation between the datasets, the correlations between SIS or EIS and the email data were re-examined, ignoring changes affecting record data types. The results of this new analysis can be seen in Figure 4.15. Compared to before, the correlation has actually worsened, indicating that the intensity of a discussion is indeed not related to the impact sets.

The direct impact of a change cannot only be quantified by the size of the SIS: The changed SLOC and the node-specific similarity measures Successor Similarity and Sub-AST TED also give an indication of how large a change is. However, even with these metrics, the hypothesis could not be proven. The evaluation of these metrics in relation to the email data is not explicitly shown in this thesis.

To better understand the relationship between the impact metrics and email data, a random sample of changes was examined manually. Changes with particularly large and small Jaccard distances and changes with particularly large numbers of messages were inspected. Detailed reports can be found in Appendix A.2. Table 4.6 provides an overview of particularly noticeable properties



(a) $r = -0.006$; $r_S = 0.071$; $\tau = 0.055$; No correlation **(b)** $r = 0.014$; $r_S = 0.017$; $\tau = 0.013$; No correlation

Figure 4.15 – Relationship between |SIS| or |EIS| and mailing list discussion intensity, ignoring changes affecting record data types

of changes and their discussion on the mailing list. First of all, it is noticeable that the impact metrics do not always reflect all changes to the source code. This may be due to the fact that the edited source code is not yet in use. However, it is more likely that the configuration for which QEMU is compiled (x86-64) does not use the code. This misestimation of change impact is due to the evaluation methodology and could be fixed by compiling QEMU for all possible configurations, which was not in the scope of this thesis. However, the resulting extremely high additional compilation effort conflicts with the requirement for efficient impact quantification.

Description	Amount
Individually examined changes	47
Not all source code changes captured by metrics	7
Changes with delayed impact	2
Discussion smaller than expected (author's opinion)	2
Discussion larger than expected (author's opinion)	5

Table 4.6 – Summary of individually examined changes

The impact of some changes is delayed and cannot be measured directly with the chosen approach. Two examples of this have been identified: One was a new framework that was introduced, which was discussed heavily at the time. However, since the new functionalities were not used until a later commit in the patch series, no major change in the GRG was detected. In another example, a function was marked as deprecated. This initially has no impact on the program behavior, but on how the software is developed in the future. Such influences are not measurable with the chosen approach, as they only affect the program after a longer period of time. Nevertheless, it is advisable and necessary for the developers to discuss such changes in detail.

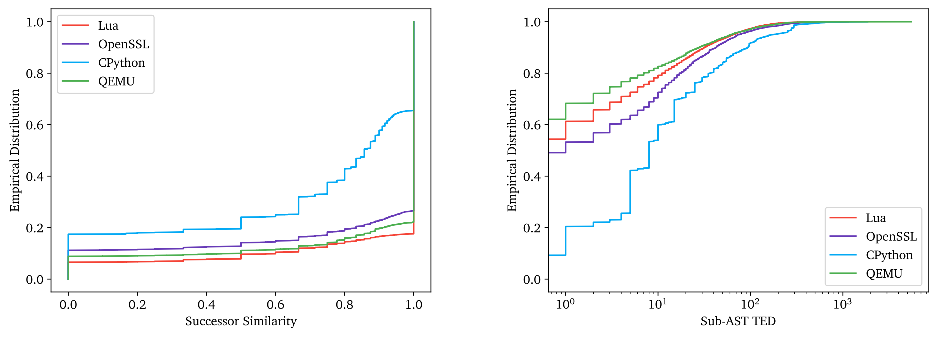
Up to this point, a number of reasons have been given for rejecting the working hypothesis that “large change impact causes large social interaction and vice versa”. Of course, this may be because there is in fact no correlation between the two variables. On the other hand, it may also be because the chosen metrics for change and social interaction do not represent these two variables correctly. Each of the change impact metrics represents only a portion of the change. EIS and SIS are actually overestimates of the parts of the program that may be impacted. Sub-AST TED, Successor Similarity, and changed SLOC reflect how large a portion of the program was actually changed. A complete and sound impact quantification would have to combine all of these measurements without over- or underestimation.

Measuring social interaction via mailing list discussion may not be ideal either. A not negligible share of the examined commits entail unexpectedly long discussions. It is doubtful that all of these discussions of this length are necessary, or whether the amount of information transferred is actually as great as the measurement suggests. Colloquially, this phenomenon is called bikeshedding, according to which less complex matters are discussed for longer than necessary because everyone can easily form their own opinion [Mcf17]. It has also been observed that changes have been submitted as part of a larger patch series, and the discussion of those commits has not been captured by PaStA because it has been linked to another commit from that patch series.

Assume that the impact quantification metrics presented correctly represent the magnitude of a change and that the intensity of the discussions has been correctly estimated after all: The developers of (open source) projects could use the metrics as a guideline for their review process. Thus, the unnecessary effort caused by bikeshedding discussions might be better spent on actually impactful changes instead.

4.5 Correlating LineImpact and Impact Metrics

Finally, the node-specific metrics of successor similarity and Sub-AST TED will be examined. In contrast to the metrics evaluated so far, which measure the potential impact of a change on the entire program, these two metrics measure the internal changes of a node in the GRG. For each change, both metrics are computed for all nodes that are directly impacted, i.e. all nodes that are contained in the SIS.



(a) For each node with successor similarity 1, the node either has not changed at all or the dependencies in the GRG stayed the same.

(b) For each node with Sub-AST TED = 0, the node has not changed internally.

Figure 4.16 – The empirical distribution of successor similarity and Sub-AST TED for different open source projects. For each project, the measurements of all nodes from all SISs across all changes were plotted.

First of all, both metrics are checked for their significance. Figure 4.16 shows the empirical distribution of successor similarity and Sub-AST TED across all nodes in all SISs across all changes. For Lua, OpenSSL and QEMU, about 78 % of all nodes in the SIS have a successor similarity = 1, so no change was detected. In contrast, only around 55 % of those nodes have a Sub-AST TED = 0.

This makes successor similarity much worse at determining whether the behavior of a node in the GRG has actually changed. To avoid any possible misunderstandings: The content of a node (function) does not really have to change for it to appear in the SIS. It is sufficient that for example a record data type changes on which this node depends, so that its local and global hash also changes. This explains the case that a node has Sub-AST TED = 0 and it still appears in the SIS.

Similar to Figure 4.10, CPython is also out of the ordinary in Figure 4.16. Nevertheless, successor similarity (34 % of nodes at 1) and Sub-AST TED (12 % of nodes at 0) behave similarly for CPython as for the other projects. Again, successor similarity performs worse in distinguishing changes from non-changes to nodes. The reason that CPython behaves so dissimilarly to the other projects is again due to the fact that part of the source code is generated automatically in CPython. This means that the proportion of modified nodes is higher overall. In the following, only Sub-AST TED will be examined in more detail as it results in a finer grained impact analysis on node level.

Sub-AST TED is in principle a similar measure of change as SLOC. Both measure an absolute amount of change to a source file. However, the SLOC measurement is less precise, since trivial changes such as new comments, empty lines, etc. are also captured. Sub-AST TED makes use of a similar property as *c*Hash: Trivial changes are not part of the AST and are therefore not counted. Figure 4.17 shows the comparison of SLOC and Sub-AST TED. In this and all following figures, Sub-AST TED is summed for all affected elements in the SIS. In some cases, Sub-AST TED is not calculated for all elements of the SIS because the Sub-AST of some functions is too large to calculate TED in a reasonable time. To avoid distorting the evaluation too much, no changes with incompletely processed Sub-AST TED are included in the plots. The doubly logarithmic plot shows a cohesion between the two variables, but Pearson's r indicates only a fair linear correlation.

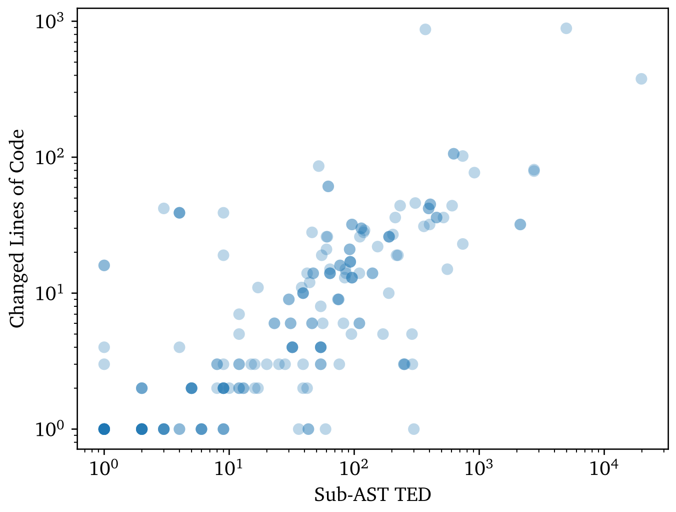
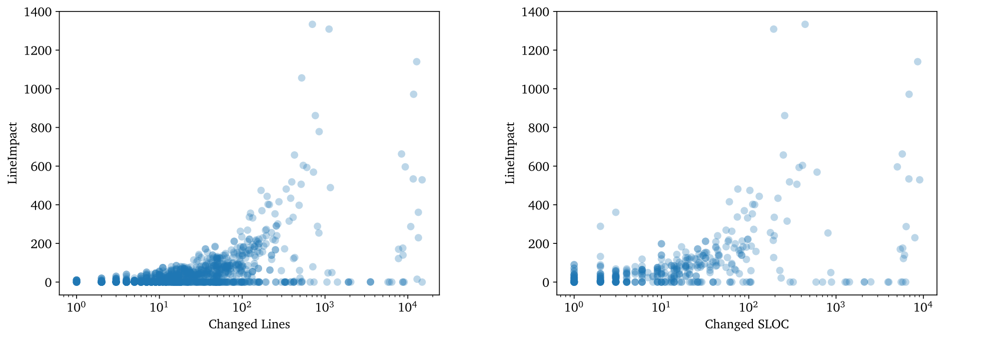


Figure 4.17 – Relationship between changed SLOC and Sub-AST TED for CPython. Changes with no impact ($|\text{SIS}| = 0$) and/or with an incomplete Sub-AST TED result are excluded. $r = 0.447$; $r_S = 0.744$; $\tau = 0.607$; Fair linear, moderate monotonic correlation

The problem is that the simple measurement of the changed SLOC contains too much noise. As already mentioned in Section 2.5, Harding therefore proposes the metric LineImpact. It filters the relevant changed lines of code by ignoring empty, duplicated, copy-pasted and churned lines, leaving only 5 % of changed lines for impact analysis [Har18]. The effect of this filtering can be seen in Figure 4.18. LineImpact is compared to all changed lines in a Git commit and the coarsely filtered SLOC, as described in Section 3.3.2. Both measurements clearly still correlate fairly well. It is however remarkable that LineImpact classifies a large amount of changes with zero impact, which is due to the strict filtering. CPython was chosen as the dataset for these analyses. LineImpact is a commercial product, therefore the impact data could not be generated freely for any open source project. The author of LineImpact kindly provided the dataset for CPython.



(a) $r = 0.339$; $r_S = 0.519$; $\tau = 0.42$; Fair correlation **(b)** $r = 0.347$; $r_S = 0.435$; $\tau = 0.379$; Fair correlation

Figure 4.18 – LineImpact in relation to changed lines and SLOC for CPython, as defined in Section 3.3.2

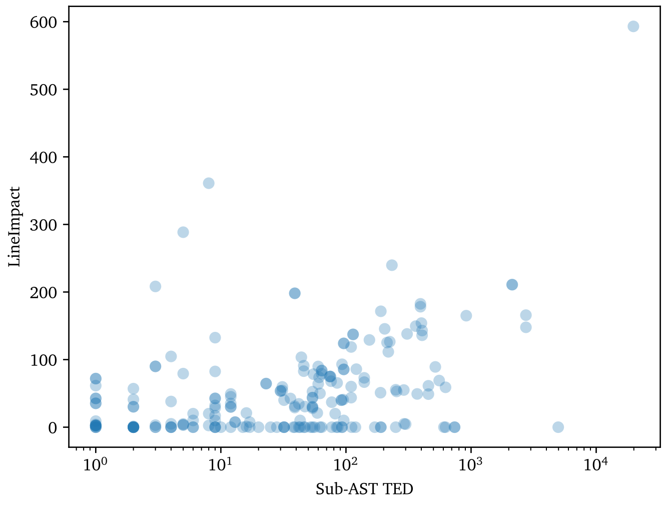


Figure 4.19 – Relationship between LineImpact and Sub-AST TED for CPython. Changes with no impact ($|\text{SIS}| = 0$) and/or with an incomplete Sub-AST TED result are excluded. $r = 0.566$; $r_S = 0.356$; $\tau = 0.267$; Moderate linear, fair monotonic correlation

The relationship between LineImpact and Sub-AST TED is shown in Figure 4.19. In contrast to the changed SLOC, the linear correlation has improved, but the monotonic correlation has worsened.

The improvement of the linear correlation is due to the fact that LineImpact also ignores trivial changes. The decrease in the monotonic correlation is caused by the many changes that have LineImpact = 0. A random selection of those changes was inspected manually, the results can be seen in Table A.5. Some of the changes actually should be rated with LineImpact = 0, since they are simply copy-paste edit operations, whereas for others the reasons for the rating cannot be understood with the help of Harding's publications. Since the source code for LineImpact is closed source, the cause of this underestimation cannot be investigated in the context of this thesis.

A perfect correlation between Sub-AST TED and LineImpact can and should not be expected, as the goals of the metrics only partially overlap. The goal of Sub-AST TED (and the other change impact metrics) is to make the impact of a change clear to a developer. In particular, if a change has a large impact even though only a few SLOC have been changed, the developer should be made aware of this. LineImpact pursues a different goal: It wants to estimate the effort that a developer had to invest for a change. That is why, among other things, copy-paste edits are weighted so low. LineImpact wants to be a non-gameable metric that allows project managers to gauge the effort of their team. In contrast to other effort estimates such as "changed lines of code" or "SCM commits per day" it should be more difficult to pretend to be working hard when measured with LineImpact [Har21]. For this use case, Sub-AST TED would be completely unsuitable, as it possibly reacts strongly to copy-paste and other effortless source code modifications.

5 CONCLUSION

In this thesis, a new change impact analysis approach based on the concept of the global reference graph was introduced. The global reference graph of a program before and after a change are combined into a diffgraph using the differential graph algorithm. From the diffgraph, the software parts directly and indirectly affected by a change, i.e., the starting and estimated impact set, can be identified by whether their local or global hash has changed. In contrast to other impact analysis approaches based on transitive closure, the diffgraph and thus the impact sets can be computed in $O(n)$, which makes this method particularly efficient. Various metrics are derived from the diffgraph that can be used to easily interpret the impact sets and quantify the change. Among all proposed metrics, Jaccard distance and Sub-AST TED have proven to be most suitable for quantifying the impact of a change on the overall structure of a program and on the behaviour of a function, respectively.

The evaluation has shown that the presented impact analysis approach can provide valuable insight into the interdependencies of a project. The metrics go beyond the traditional impact analysis goals of estimating accurate impact sets and can help a developer with interpreting the possible consequences of a proposed change. As already mentioned, the impact quantification can also serve as an indicator for guiding review discussions. Compared to the LineImpact metric, Sub-AST TED delivers similar results with a linear correlation coefficient $r = 0.566$. However, it would be advisable to repeat the measurements for social interaction and LineImpact with additional datasets for other open source projects in order to further validate and solidify the conclusions drawn from the analyses.

Change impact analysis based on the GRG and diffgraph meets all four essential requirements for CIA defined by ISO/IEC [ISO06]: It detects all possible ripple effects on other software modules, since it is based on the complete *c*Hash approach. It is able to determine the level of testing required, since it is based on the TASTING approach, which correctly identifies the set of automated software tests to be reexecuted after a change [LDL21]. In addition, the amount of evaluation required is determined by the impact quantification metrics. This is closely coupled to the third requirement, the presented CIA approach estimates the size and magnitude of a modification. Lastly, it can be said that the approach takes into account the development history of the projects studied. Even more, some metrics, such as Jaccard distance, only become valuable if the development history is taken into account, as they need a context to properly classify changes.

In addition, the presented CIA approach also has some features that are not strictly required by ISO/IEC: The impact analysis is simple and efficient, as it can be seamlessly integrated into the build processes of most C projects. The evaluation shows that the performance penalty due to impact analysis during the compilation process is bearable in most projects, especially when compared to other impact analysis approaches: In the worst case, Lua, compilation and impact quantification triple the buildtimes to 3.4 s compared to a clean build. However, this is still quicker than simply computing the transitive closure, which on its own already takes 3.75 s. For larger Projects like CPython, this difference becomes even more apparent. The additional overhead of impact analysis can possibly be compensated for by taking advantage of the accelerated compilation and reduced test times provided by *c*Hash and TASTING.

In the future, impact quantification could be extended to more programming languages than C. The concept of the GRG can be applied to any programming language that can be represented by an AST. However, extending to more languages is not the responsibility of the IA approach, but rather of *c*Hash, which would be desirable to have available for more programming languages anyway. Other impact quantification such as LineImpact is mostly language independent, which cannot be achieved in this case without rewriting *c*Hash for each language.

A weakness of the GRG for impact analysis is that record data types and preprocessor macros are not part of the graph. Therefore, especially the SIS is in many cases an overestimation. It would be easy to include structs in the GRG, since only `chash-global` would have to be adjusted. However, there are also approaches on how not to lose the information about macros in the preprocessing stage.[23]

The impact metrics specifically for QEMU suffered from the fact that only a small part of the software was studied, since code that does not contribute to the GRG is excluded from IA. Therefore, in the future, it remains to be explored how to apply impact quantification to highly configurable systems.

I would like to take this opportunity to thank Ralf Ramsauer for his advice and support on obtaining the email dataset from PaStA. I also want to thank Bill Harding, who quickly and without much fuss provided me with the dataset on LineImpact. My thanks finally go to Tobias Landsberg, who, as the supervisor of this thesis, always quickly had a good answer to all my questions.

[23]Pending thesis at the Institute for Systems Engineering: “Preprocessed information: Extend the C preprocessor with source code markers”: `https://www.sra.uni-hannover.de/Theses/2021/BA-ARA-cpp.html`

APPENDIX A

A.1 Commit Series

Project	Commit Range
QEMU	`4250da10..1d806cef`
CPython	-
OpenSSL	`dceb99a5..f123043f`
Lua	`8d9ea59d..439e45a2`

Table A.1 – Examined commit series for each project. The Git-DAGs were extracted using `git log --pretty='format:%H %P' <commit-range>`

CPython employs a rebase-oriented workflow, therefore it's commit history is in constant change. The compiled commits were not selected from a commit range, but rather from the commits that were available in the LineImpact dataset and the currently cloned CPython repository at the same time.

A.2 Comments on Changes

Commit ID	Mails	Sub-AST TED	Jaccard Distance	Kind of Change	Comment on Mails	Other
7b733862	0	6301	0.519	split main into three subfunctions	large discussion expected, found only one message	
4749079c	3	477	0.535	changes to structs, new & modified functions, new macros	short discussion, but rest of thread has longer discussion	high impact is expected
2880ffb0	4	22	0.482	new global variables, incorporated in some functions	low discussion expected	
164c374b	4	64	0.521	refactor object-freeing function	part of larger patch series (26). could not find replies to initial message	this function is heavily used, but hidden behind multiple layers of abstraction. prime example for surprisingly impactful function
269bd5d8	4	0	0.511	change member of often used struct, touch files for many different targets	part of large patch series (39), possibly longer discussion in other subthreads	
b56668bb	5	0	0.520	removal of unused struct member, adapt some functions to this	low discussion expected, since it is an unused member	we only track the struct changing, not the function. this is caused by only building for one specific configuration, where the function is likely unused. potential weakness for cia approach (but is to be expected)
7b3cb803	6	160	0.517	refactor multiple frequently used functions	number of mails seems fitting	
cbf97d5b	7	0	0.504	modify Python-C-code generation of an often used template	number of mails seems fitting	
9c09a251	7	111	0.519	modify CPUState struct and functions that use it	medium discussion expected and found	some modified functions appear to be missing from the built configuration and therefore are not measured with Sub-AST TED
e957ad8a	9	14	0.506	refactor a function, only a stylistic change	relatively high discussion unexpected, as it does not look like a large change (semantically). could not find discussion online	
0c0fcc20	10	156	0.525	modify global CPUState struct, modify some functions	amount of mails looks ok, probably none missed	again, some functions do not appear in the impact set. modification to cpustate likely responsible for high Jaccard distance
a26fc6f5	11	21	0.516	changes to struct MemTxAttrs and some functions using it	seems to have found the correct discussion	high Jaccard distance likely caused by change to struct. change to function not detected at all, but function has changed in other commits → this function was not part of the target in this commit, but in others it was
2f3a57ee	13	0	0.493	modifications to struct and functions	high discussion expected (and found)	Sub-AST TED metric shows nothing, since only really modified function does not appear to be used in this configuration. high Jaccard distance is caused by CPUState struct
22dc8663	15	16	0.484	really simple modification to one function	discussion seems too large for this simple change. other patches in this series seem to be in response to this first patch, so this might skew the data	modified function is used in many places, this explains high Jaccard distance

Table A.2 – QEMU. Comments on a random selection of commits with high Jaccard distance

Commit ID	Mails	Sub-AST TED	Jaccard Distance	Kind of Change	Comment on Mails	Other
`e1d322c4`	0	0	0.008	change version tag	no discussion required	
`0f0998f6`	1	35	0.057	minor function refactoring	small discussion expected	
`c38c1c14`	2	16	0.008	minor function modification		
`420ae1fc`	3	502	0.023	refactor by pulling out code into static functions	not much discussion expected	
`82e870ba`	3	37	0.001	refactor iterator in function	amount of mails seems ok	
`d8fa8442`	4	27	0.018	refactor magic numbers with getters	amount of mails seems ok	
`712f807e`	4	0	0.004	revert a minor commit	could not find messages online	
`7ffcb73d`	5	0	0.001	move around headers	discussion seems too large for this minor change	
`a857d91d`	5	16	0.020	change multiple debug macro calls	discussion seems ok, since many files were touched	interestingly, many changed functions do not lead to a high Jaccard distance, contrary to a single struct
`38841dcd`	6	82	0.018	add new initializer function	discussion seems ok	
`c3e95551`	7	419	0.001	move parser function to other module	discussion seems ok, maybe even a bit large	high Sub-AST TED: new function & heavily modified function; low Jaccard distance: parsing happens near main, low impact set
`6fd5bef1`	8	158	0.029	change return value of void functions to bool (error-value)	number of messages seems ok. however, could not find discussion thread online	
`848c7092`	8	308	0.009	refactoring, replace occurrences of one function with new function	could find discussion	
`d7741743`	9	1	0.001	change of struct and one invoking function	could not find much discussion around this commit online	this struct is less frequently used
`36adac49`	11	62	0.001	change struct and functions using it	high discussion expected and found	this struct seems to have a small impact on the rest of the system
`4db6ceb0`	13	24	0.009	refactor semantics of status flags	discussion found	semantic change warrants longer discussion, even if the measured impact is low
`645ae7d8`	17	136	0.019	refactor: return string in allocated buffer instead of provided buffer	could not find the discussion online	high Sub-AST TED, since many functions are touched. however, the discussion seems too large for this change
`4bb4a273`	32	6	0.001	deprecate a function	could not find the related discussion	deprecation has very low impact, discussion is too large

Table A.3 – QEMU. Comments on a random selection of commits with small Jaccard distance

Commit ID	Mails	Sub-AST TED	Jaccard Distance	Kind of Change	Comment on Mails	Other
`46517dd4`	41	2	0.038	move around includes, so recompilations happen less frequently. many files touched!	large discussion was found	discussion mostly around breaking build chains
`a783f8ad`	42	0	0.001	add framework for later change, that will use the framework	could not find discussion online	
`e4d7019e`	43	29	0.019	small fix	could not find the large discussion	
`80f5c011`	43	18	0.019	small fix, changes to testing	could find some discussion	discussion mostly revolves around tests
`cdf80365`	43	7	0.002	deprecation, small change to code	could find large discussion	discussion revolves around semantic change of deprecation
`862b4a29`	47	6	0.047	large change: add display emulation module	large discussion seems ok	changes are not part of compilation target, therefore the change was measured as being small
`e4ec5ad4`	49	95	0.193	add replay behaviour, touch many files	large discussion found	discussion and jaccard seem to match
`c8bb23cb`	51	341	0.028	add some functions, changes to struct	large discussion found	large discussion around new functions
`bd227060`	59	60	0.020	performance optimizations	large discussion found	
`8de702cb`	60	2	0.000	add capabilities to read characters from console when semihosting	could not find discussion online	most of the new code not compiled in selected configuration
`119906af`	60	109	0.015	changes to mmap	many messages found	mmap seems to be a sensitive topic
`9b12dfa0`	76	540	0.012	add new command line option, add large function to support it	large discussion expected and found	
`557d2bdc`	80	604	0.021	add encryption key management, many new and changed functions	large discussion expected and found	
`b8968c87`	80	196	0.025	add functionality concerning bitmaps	many messages found	
`0b6786a9`	122	212	0.019	some functions refactored	could not find a large discussion online	number of mails seems too high

Table A.4 – QEMU. Comments on a random selection of commits with many related emails

Commit ID	Sub-AST TED	LineImpact	Kind of Change	Comment
`1b940eb4`	0	0.0	Mark function as extern	No impact expected from both metrics
`d80f4265`	2	0.0	Fix escape characters in strings	
`16ef0f9f`	2	0.0	Increase version tag	LineImpact ok
`489699ca`	4	0.0	Some refactoring	LineImpact lower than expected
`1a672a59`	9	0.0	Cast variable to void	LineImpact ok
`823460da`	20	0.0	Fix function behaviour	LineImpact lower than expected
`fe847a62`	32	0.0	Move code	LineImpact ok
`298ee657`	46	0.0	Fix function behaviour	LineImpact lower than expected
`00710e63`	62	0.0	Multiple macro flags updated	LineImpact should be low due to copy&paste, but not 0
`3f81e962`	93	0.0	Fix function behaviour	LineImpact lower than expected
`ac8f72cd`	190	0.0	Multiple similar new macro uses	LineImpact should be low due to copy&paste, but not 0
`5644c7b3`	738	0.0	Change function signatures, move code	LineImpact lower than expected

Table A.5 – CPython. Comments on a random selection of commits with LineImpact = 0

LIST OF ACRONYMS

AIS	Actual Impact Set
AP-TED	All Path Tree Edit Distance
AST	Abstract Syntax Tree
BFS	Breadth-First Search
CIA	Change Impact Analysis
CRT	Cross-Reference Table
DAG	Directed Acyclic Graph
DFS	Depth-First Search
EIS	Estimated Impact Set
FNIS	False Negative Impact Set
FPIS	False Positive Impact Set
GED	Graph Edit Distance
GRG	Global Reference Graph
IA	Impact Analysis
IS	Impact Set
JSON	JavaScript Object Notation
NLP	Natural Language Processing
PaStA	Patch Stack Analysis
RTED	Robust Tree Edit Distance
RTS	Regression-Test Selection
SCM	Source Code Management
SIS	Starting Impact Set
SLOC	Source Lines of Code
SUT	Software-Under-Test
TED	Tree Edit Distance

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

REFERENCES

[AB93] Robert S Arnold and Shawn A Bohner. “Impact Analysis - Towards A Framework for Comparison.” In: *1993 Conference on Software Maintenance*. IEEE, 1993, pp. 292–301.

[Ako18] Haldun Akoglu. “User’s guide to correlation coefficients.” en. In: *Turkish Journal of Emergency Medicine* 18.3 (Sept. 2018), pp. 91–93. ISSN: 24522473. DOI: 10.1016/j.tjem.2018.08.001. URL: https://linkinghub.elsevier.com/retrieve/pii/S2452247318302164 (visited on 08/26/2021).

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, principles, techniques, and tools*. Reading, Mass: Addison-Wesley Pub. Co, 1986. ISBN: 978-0-201-10088-4.

[BBSY05] L. Badri, M. Badri, and D. St-Yves. “Supporting predictive change impact analysis: a control call graph based technique.” In: *12th Asia-Pacific Software Engineering Conference (APSEC’05)*. Taipei, Taiwan: IEEE, 2005, 9 pp. ISBN: 978-0-7695-2465-8. DOI: 10.1109/APSEC.2005.100. URL: http://ieeexplore.ieee.org/document/1607149/ (visited on 08/04/2021).

[BJG09] Jørgen Bang-Jensen and Gregory Z. Gutin. *Digraphs*. Springer Monographs in Mathematics. London: Springer London, 2009. ISBN: 978-0-85729-041-0 978-1-84800-998-1. DOI: 10.1007/978-1-84800-998-1. URL: http://link.springer.com/10.1007/978-1-84800-998-1 (visited on 09/13/2021).

[Bla08] Sue Black. “Deriving an approximation algorithm for automatic computation of ripple effect measures.” en. In: *Information and Software Technology* 50.7-8 (June 2008). Number: 7-8, pp. 723–736. ISSN: 09505849. DOI: 10.1016/j.infsof.2007.07.008. URL: https://linkinghub.elsevier.com/retrieve/pii/S0950584907000791 (visited on 05/11/2021).

[BM07] Olga Baysal and Andrew J. Malton. “Correlating Social Interactions to Release History during Software Evolution.” In: *Fourth International Workshop on Mining Software Repositories (MSR’07:ICSE Workshops 2007)*. Minneapolis, MN, USA: IEEE, May 2007, pp. 7–7. ISBN: 978-0-7695-2950-9. DOI: 10.1109/MSR.2007.4. URL: http://ieeexplore.ieee.org/document/4228644/ (visited on 09/16/2021).

[CCDP07] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. “Identifying Changed Source Code Lines from Version Repositories.” In: *Fourth International Workshop on Mining Software Repositories (MSR’07:ICSE Workshops 2007)*. Minneapolis, MN: IEEE, May 2007, pp. 14–14. ISBN: 978-0-7695-2950-9. DOI: 10.1109/MSR.2007.14. URL: https://ieeexplore.ieee.org/document/4228651/ (visited on 09/28/2021).

[CDR09] Michel Chilowicz, Etienne Duris, and Gilles Roussel. "Syntax tree fingerprinting for source code similarity detection." In: *2009 IEEE 17th International Conference on Program Comprehension*. Vancouver, BC, Canada: IEEE, May 2009, pp. 243–247. ISBN: 978-1-4244-3998-0. DOI: 10.1109/ICPC.2009.5090050. URL: http://ieeexplore.ieee.org/document/5090050/ (visited on 09/24/2021).

[Die17] Reinhard Diestel. *Graph Theory*. 5th ed. 2017. Graduate Texts in Mathematics 173. Berlin, Heidelberg: Springer Berlin Heidelberg : Imprint: Springer, 2017. ISBN: 978-3-662-53622-3. DOI: 10.1007/978-3-662-53622-3.

[Die+17] Christian Dietrich et al. "cHash: Detection of Redundant Compilations via AST Hashing." In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, July 2017, pp. 527–538. ISBN: 978-1-931971-38-6. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/dietrich.

[DLFO08] Andrea De Lucia, Fausto Fasano, and Rocco Oliveto. "Traceability management for impact analysis." In: *2008 Frontiers of Software Maintenance*. Beijing, China: IEEE, Sept. 2008, pp. 21–30. ISBN: 978-1-4244-2654-6. DOI: 10.1109/FOSM.2008.4659245. URL: http://ieeexplore.ieee.org/document/4659245/ (visited on 07/27/2021).

[ETL14] Jon Eyolfson, Lin Tan, and Patrick Lam. "Correlations between bugginess and time-based commit characteristics." en. In: *Empirical Software Engineering* 19.4 (Aug. 2014), pp. 1009–1039. ISSN: 1382-3256, 1573-7616. DOI: 10.1007/s10664-013-9245-0. URL: http://link.springer.com/10.1007/s10664-013-9245-0 (visited on 10/21/2021).

[FCX13] Jianglang Feng, Baojiang Cui, and Kunfeng Xia. "A Code Comparison Algorithm Based on AST for Plagiarism Detection." In: *2013 Fourth International Conference on Emerging Intelligent Data and Web Technologies*. Xi'an, Shaanxi, China: IEEE, Sept. 2013, pp. 393–397. ISBN: 978-1-4799-2141-6 978-1-4799-2140-9. DOI: 10.1109/EIDWT.2013.73. URL: http://ieeexplore.ieee.org/document/6631651/ (visited on 09/24/2021).

[FRB17] Andreas Fischer, Kaspar Riesen, and Horst Bunke. "Improved quadratic time approximation of graph edit distance by combining Hausdorff matching and greedy assignment." en. In: *Pattern Recognition Letters* 87 (Feb. 2017), pp. 55–62. ISSN: 01678655. DOI: 10.1016/j.patrec.2016.06.014. URL: https://linkinghub.elsevier.com/retrieve/pii/S0167865516301386 (visited on 05/06/2021).

[Fü17] Ludwig Füracker. "Effiziente globale Änderungsanalysen in großen C-Projekten durch Syntaxbaum-Hashing im Clang Compiler." Master's Thesis. Dept. of Computer Science: University of Erlangen, Oct. 2017.

[GHR09] Daniel M. German, Ahmed E. Hassan, and Gregorio Robles. "Change impact graphs: Determining the impact of prior codechanges." en. In: *Information and Software Technology* 51.10 (Oct. 2009), pp. 1394–1408. ISSN: 09505849. DOI: 10.1016/j.infsof.2009.04.018. URL: https://linkinghub.elsevier.com/retrieve/pii/S095058490900069X (visited on 08/04/2021).

[Har18] William Bates Harding. *Count lines of code, but don't be naive about what you're really measuring*. 2018. URL: https://www.gitclear.com/count_lines_of_code_but_dont_be_naive (visited on 10/23/2021).

[Har21] William Bates Harding. "Software effort estimates vs popular developer productivity metrics: case study of empirical correlation." In: *ESEC/FSE 2021: manuscript submitted for publication*. Feb. 2021. URL: https://api.amplenote.com/v2/view/aMM59VVrwALefeHoPbJE2otR/attachments/3ff02c17-aebe-47e6-a1a1-7195d42a0653 (visited on 08/18/2021).

[Hat+08] Lile Hattori et al. "On the Precision and Accuracy of Impact Analysis Techniques." In: *Seventh IEEE/ACIS International Conference on Computer and Information Science (icis 2008)*. IEEE, May 2008, pp. 513–518. ISBN: 978-0-7695-3131-1. DOI: 10.1109/ICIS.2008.104. URL: http://ieeexplore.ieee.org/document/4529870/ (visited on 08/03/2021).

[ISO06] ISO/IEC. *14764:2006: Software Engineering - Software Life Cycle Processes - Maintenance*. 2006. URL: https://www.iso.org/standard/39064.html (visited on 07/22/2021).

[Jac12] Paul Jaccard. "The Distribution of the Flora in the Alpine Zone." en. In: *New Phytologist* 11.2 (Feb. 1912), pp. 37–50. ISSN: 0028-646X, 1469-8137. DOI: 10.1111/j.1469-8137.1912.tb05611.x. URL: http://doi.wiley.com/10.1111/j.1469-8137.1912.tb05611.x (visited on 05/10/2021).

[LDL21] Tobias Landsberg, Christian Dietrich, and Daniel Lohmann. "TASTING: Reuse Test-Case Execution by Global AST Hashing." In: *Work in progress: manuscript to be submitted for publication*. 2021.

[Li+13] Bixin Li et al. "A survey of code-based change impact analysis techniques." en. In: *Software Testing, Verification and Reliability* 23.8 (Dec. 2013). Number: 8, pp. 613–646. ISSN: 09600833. DOI: 10.1002/stvr.1475. URL: http://doi.wiley.com/10.1002/stvr.1475 (visited on 05/10/2021).

[Mcf17] Paul Mcfedries. "Agile words [Technically Speaking]." In: *IEEE Spectrum* 54.6 (June 2017), pp. 21–21. ISSN: 0018-9235, 1939-9340. DOI: 10.1109/MSPEC.2017.7934225. URL: https://ieeexplore.ieee.org/document/7934225/ (visited on 10/22/2021).

[MO05] Robert William Mecklenburg and Andrew Oram. *Managing projects with GNU make*. 3rd ed. Beijing ; Cambridge [Mass.]: O'Reilly, 2005. ISBN: 978-0-596-00610-5.

[MOH07] Kjetil Molokken-Ostvold and Nils Christian Haugen. "Combining Estimates with Planning Poker–An Empirical Study." In: *2007 Australian Software Engineering Conference (ASWEC'07)*. ISSN: 1530-0803. Melbourne, Australia: IEEE, Apr. 2007, pp. 349–358. ISBN: 978-0-7695-2778-9. DOI: 10.1109/ASWEC.2007.15. URL: http://ieeexplore.ieee.org/document/4159687/ (visited on 10/19/2021).

[Ngu+07] Vu Nguyen et al. "A SLOC Counting Standard." In: *COCOMO II Forum 2007*. 2007.

[OHK93] A.Jefferson Offutt, Mary Jean Harrold, and Priyadarshan Kolte. "A software metric system for module coupling." en. In: *Journal of Systems and Software* 20.3 (Mar. 1993), pp. 295–308. ISSN: 01641212. DOI: 10.1016/0164-1212(93)90072-6. URL: https://linkinghub.elsevier.com/retrieve/pii/0164121293900726 (visited on 10/14/2021).

[PA11] Mateusz Pawlik and Nikolaus Augsten. "RTED: A Robust Algorithm for the Tree Edit Distance." In: *arXiv:1201.0230 [cs]* (Dec. 2011). arXiv: 1201.0230. URL: http://arxiv.org/abs/1201.0230 (visited on 05/06/2021).

[PA15] Mateusz Pawlik and Nikolaus Augsten. “Efficient Computation of the Tree Edit Distance.” en. In: *ACM Transactions on Database Systems* 40.1 (Mar. 2015). Number: 1, pp. 1–40. ISSN: 0362-5915, 1557-4644. DOI: 10.1145/2699485. URL: https://dl.acm.org/doi/10.1145/2699485 (visited on 05/06/2021).

[PA16] Mateusz Pawlik and Nikolaus Augsten. “Tree edit distance: Robust and memory-efficient.” en. In: *Information Systems* 56 (Mar. 2016), pp. 157–173. ISSN: 03064379. DOI: 10.1016/j.is.2015.08.004. URL: https://linkinghub.elsevier.com/retrieve/pii/S0306437915001611 (visited on 05/06/2021).

[Pim17] João Felipe N. Pimentel. *Python APTED algorithm for the Tree Edit Distance*. 2017. URL: https://github.com/JoaoFelipe/apted (visited on 09/30/2021).

[RLM16] Ralf Ramsauer, Daniel Lohmann, and Wolfgang Mauerer. “Observing Custom Software Modifications: A Quantitative Approach of Tracking the Evolution of Patch Stacks.” en. In: *Proceedings of the 12th International Symposium on Open Collaboration*. Berlin Germany: ACM, Aug. 2016, pp. 1–4. ISBN: 978-1-4503-4451-7. DOI: 10.1145/2957792.2957810. URL: https://dl.acm.org/doi/10.1145/2957792.2957810 (visited on 10/18/2021).

[RLM19] Ralf Ramsauer, Daniel Lohmann, and Wolfgang Mauerer. “The List is the Process: Reliable Pre-Integration Tracking of Commits on Mailing Lists.” In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. Montreal, QC, Canada: IEEE, May 2019, pp. 807–818. ISBN: 978-1-72810-869-8. DOI: 10.1109/ICSE.2019.00088. URL: https://ieeexplore.ieee.org/document/8812060/ (visited on 10/18/2021).

[RMT09] Mehwish Riaz, Emilia Mendes, and Ewan Tempero. “A systematic review of software maintainability prediction and metrics.” In: *2009 3rd International Symposium on Empirical Software Engineering and Measurement*. Lake Buena Vista, FL, USA: IEEE, Oct. 2009, pp. 367–377. ISBN: 978-1-4244-4842-5. DOI: 10.1109/ESEM.2009.5314233. URL: http://ieeexplore.ieee.org/document/5314233/ (visited on 10/19/2021).

[RT] Joel Rosdahl and Andrew Tridgell. *Ccache – a fast C/C++ compiler cache*. URL: https://ccache.dev/ (visited on 09/15/2021).

[RY20] Xavier Rival and Kwangkeun Yi. *Introduction to static analysis an abstract interpretation perspective*. English. OCLC: 1206371988. 2020. ISBN: 978-0-262-35665-7. URL: http://public.eblib.com/choice/PublicFullRecord.aspx?p=6388577 (visited on 08/02/2021).

[Sag+06] Tobias Sager et al. “Detecting similar Java classes using tree algorithms.” en. In: *Proceedings of the 2006 international workshop on Mining software repositories - MSR ’06*. Shanghai, China: ACM Press, 2006, p. 65. ISBN: 978-1-59593-397-3. DOI: 10.1145/1137983.1138000. URL: http://portal.acm.org/citation.cfm?doid=1137983.1138000 (visited on 10/27/2021).

[Sch12] Rainer Schlittgen. *Einführung in die Statistik: Analyse und Modellierung von Daten*. ger. 12., korrigierte Aufl. Lehr- und Handbücher der Statistik. München: Oldenbourg, 2012. ISBN: 978-3-486-71591-0 978-3-486-71524-8.

[Sch87] N.F. Schneidewind. “The State of Software Maintenance.” In: *IEEE Transactions on Software Engineering* SE-13.3 (Mar. 1987), pp. 303–310. ISSN: 0098-5589. DOI: 10.1109/TSE.1987.233161. URL: http://ieeexplore.ieee.org/document/1702216/ (visited on 07/22/2021).

[Ser19] Francesc Serratosa. “Graph edit distance: Restrictions to be a metric.” en. In: *Pattern Recognition* 90 (June 2019), pp. 250–256. ISSN: 00313203. DOI: 10.1016/j.patcog.2019.01.043. URL: https://linkinghub.elsevier.com/retrieve/pii/S0031320319300639 (visited on 05/06/2021).

[Tao09] Terence Tao. *Benford’s law, Zipf’s law, and the Pareto distribution*. 2009. URL: https://terrytao.wordpress.com/2009/07/03/benfords-law-zipfs-law-and-the-pareto-distribution/ (visited on 10/29/2021).

[Ton03] P. Tonella. “Using a concept lattice of decomposition slices for program understanding and impact analysis.” en. In: *IEEE Transactions on Software Engineering* 29.6 (June 2003). Number: 6, pp. 495–509. ISSN: 0098-5589. DOI: 10.1109/TSE.2003.1205178. URL: http://ieeexplore.ieee.org/document/1205178/ (visited on 05/12/2021).

[War75] Henry S. Warren. “A modification of Warshall’s algorithm for the transitive closure of binary relations.” en. In: *Communications of the ACM* 18.4 (Apr. 1975), pp. 218–220. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/360715.360746. URL: https://dl.acm.org/doi/10.1145/360715.360746 (visited on 09/14/2021).

[Wu06] Jingwei Wu. “Open Source Software Evolution and Its Dynamics.” PhD Thesis. UWSpace, 2006. URL: http://hdl.handle.net/10012/1095.

[Zha+15] Ying Zhang et al. “ABC: Accelerated Building of C/C++ Projects.” In: *2015 Asia-Pacific Software Engineering Conference (APSEC)*. New Delhi: IEEE, Dec. 2015, pp. 182–189. ISBN: 978-1-4673-9644-8. DOI: 10.1109/APSEC.2015.27. URL: http://ieeexplore.ieee.org/document/7467299/ (visited on 08/18/2021).