Improving Software Dependability through Documentation Analysis

by

Edmund Wong

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Engineering

Waterloo, Ontario, Canada, 2019

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

Internal-External Examiner:	Meiyappan Nagappan Assistant Professor, Dept. of Computer Science, University of Waterloo
Supervisor:	Lin Tan Associate Professor, Dept. of Electrical and Computer Engineering, University of Waterloo
Internal Member:	Arie Gurfinkel Associate Professor, Dept. of Electrical and Computer Engineering, University of Waterloo
Internal Member:	Mahesh Tripunitara Associate Professor, Dept. of Electrical and Computer Engineering, University of Waterloo
External Member:	Nicholas A. Kraft Software Researcher, ABB Corporate Research

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Software documentation contains critical information that describes a system's functionality and requirements. Documentation exists in several forms, including code comments, test plans, manual pages, and user manuals. The lack of documentation in existing software systems is an issue that impacts software maintainability and programmer productivity. Since some code bases contain a large amount of documentation, we want to leverage these existing documentation to improve software dependability. Specifically, we utilize documentation to help detect software bugs and repair corrupted files, which can reduce the number of software error and failure to improve a system's reliability (e.g., continuity of correct service). We also generate documentation (e.g., code comment) automatically to help developers understand the source code, which helps improve a system's maintainability (e.g., ability to undergo repairs and modifications).

In this thesis, we analyze software documentation and propose two branches of work, which focuses on three types of documentation including manual pages, code comments, and user manuals. The first branch of work focuses on documentation analysis because documentation contains valuable information that describes the behavior of the program. We automatically extract constraints from documentation and apply them on a dynamic analysis symbolic execution tool to find bugs in the target software, and we extract constraints manually from documentation and apply them on a structured-file parsing application to repair corrupted PDF files. The second branch of work focuses on automatic code comment generation to improve software documentation.

For documentation analysis, we propose and implement DASE and DocRepair. DASE leverages automatically extracted constraints from documentation to improve a dynamic analysis symbolic execution tool. DASE guides symbolic execution to focus the testing on execution paths that execute a program's core functionalities using constraints learned from the documentation. We evaluated DASE on 88 programs from five mature real-world software suites to detect software bugs. DASE detects 12 previously unknown bugs that symbolic execution would fail to detect when given no input constraints, 6 of which have been confirmed by the developers.

In DocRepair we perform an empirical study to study and repair corrupted PDF files. We create the first dataset of 319 corrupted PDF files and conduct an empirical study on 119 real-world corrupted PDF files to study the common types of file corruption. Based on the result of the empirical study we propose a technique called DocRepair. DocRepair's repair algorithm includes seven repair operators that utilizes manually extracted constraints from documentation to repair corrupted files. We evaluate DocRepair against three common PDF repair tools. Amongst the 1,827 collected corrupted files from over two corpora

of PDF files, DocRepair can successfully repair 354 files compared to Mutool, PDFtk, and GhostScript which repair 508, 41 and 84 respectively. We also propose a technique to combine multiple repair tools called DocRepair+, which can successfully repair 751 files.

In the case where there is a lack of documentation, DASE and DocRepair+ would not work. Therefore, we propose automated documentation generation to address the issue. We propose and implement CloCom+ to generate code comments by mining both existing software repositories in GitHub and a Question and Answer site, Stack Overflow. CloCom+ generated 442 unique comments for 16 Java projects. Although CloCom+ improves on previous work, SumSlice, on automatic comment generation, the quality (evaluated on completeness, conciseness, expressiveness, and usefulness) and yield (number of generated comments) are still rather low which makes the technique not ready for real-world usage.

In the future, it may be possible to combine the two proposed branches of work (documentation analysis and documentation generation) to further improve software dependability. For example, we can extract constraints from the automatically generated documentation (e.g., code comments).

Acknowledgements

I would like to thank all the little people who made this possible.

Dedication

This is dedicated to the one I love.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software documentation is written text or illustration that describes a software system. It contains information that describes a system's requirements and specifications and exists in several forms, including code comments, test plans, manual pages (also known as man pages), and user manuals (e.g., file format specification). Software maintainers rely on source code and code comment as two of the most important documentation artifacts to understand software systems [2]. Documentation had also been previously shown to improve software maintainability [3] and programmer productivity [4]. Since documentation contains informal information that describes a software system, this thesis attempts to formalize the information to obtain constraints that describe the behavior of the system. The constraints that are extracted from documentation can be used to improve a system's dependability [5], which is defined as a measure of a system's reliability, maintainability, safety, confidentiality, integrity, and availability. Specifically, this thesis focuses on utilizing documentation to detect software bugs and repair corrupted files to improve a system's reliability (e.g., continuity of correct service), and automatically generate documentation to improve a system's maintainability (e.g., ability to undergo repairs and modifications).

Code bases often contain a vast amount of documents. For example, GitHub [6] contains a total of 17 million lines of Java comments and 42 million lines of code in 1,005 Java projects [7]. Documentation contains valuable information that describes the behavior of the program. Previous work had utilized code comments for inconsistency detection between comment and code [8], and inferring method specifications from natural language API descriptions [9]. For example, a code comment from the Linux Kernel, "This function must not be called from interrupt context," specifies that the target function that the comment is describing must not be called from interrupt context [8]. Therefore, the first goal of this thesis is to analyze software documentation (including manual pages, code comments,

and user manuals) to improve software reliability. Specifically, we want to minimize the number of failures in a software system by detecting software bugs and repairing corrupted files through the use of documentation constraints. This thesis examines three types of software documentation, including manual pages (help text for Linux/Unix commands, system calls, file format etc.), source code comments, and user manuals (a technical document can contain text, images, tables, etc.). We extract constraints from the documentation, and we utilize them to perform automated bug detection and automated file repair.

Many software projects often do not contain sufficient documentation in practice [10]. Documentation can be scarce, incorrect [11], or outdated in practice [2, 12]. Therefore, the second goal of this thesis is to generate documentation automatically. Specifically, it leverages two sources of information including software repositories and question and answer (Q&A) sites for automatic documentation generation.

The theme of this thesis considers both documentation analysis and documentation generation as the two main goals. Documentation analysis relies on the existence of high-quality documentation. However, high-quality documentation may not always be available, which motivates us to propose techniques to generate them automatically. This thesis proposes techniques to improve both areas. In the future, it may be possible to take the automatically generated comments as the input for documentation analysis.

1.1 Automatic Documentation Analysis

Documentation analysis focuses on improving software reliability. We propose and implement two pieces of work for automatic documentation analysis.

First, we propose DASE to automatically detect software bugs. Software bugs can cause severe issues to the end user such as program crashes and incorrect behavior. DASE can detect software bugs automatically which helps developers to discover and fix software bugs early during the software development cycle and reduce the number of software errors and failures.

Second, we propose DocRepair to study corrupted PDF files and use that knowledge to repair corrupted PDF files with manually extracted documentation constraints. Since corrupted PDF files can cause error and failures on the target application (e.g., inability to open and display the content of the file), DocRepair repairs the corrupted files to mitigate the software failure, which helps improve software reliability.

1.1.1 General Symbolic Execution-Based Software Testing

We implement DASE to leverage documentation to help improve automated test generation and bug detection. Real world programs contain an enormous number of execution paths. Given a limited amount of time, testing tools have to prioritize the execution paths to ensure effective testing. We want to automate the process by extracting input constraints from the documentation. DASE extracts and applies grammar-based constraints on complex program inputs to help find bugs in structured-file parsing applications.

DASE guides symbolic execution to focus testing execution paths that implement program's core functionalities using constraints learned from the documentation. It analyzes two types of documentation, source code comments, and manual pages. It extracts two types of constraints, file format constraints, and valid-option constraints.

Since input constraints commonly exist in documents, symbolic execution techniques can potentially take advantage of the constraints automatically. For example, `rm` (version 6.10) only accepts 11 options including `-r` and `-f`, and `readelf` requires its input files to follow Executable and Linkable Format (ELF). Focusing on the valid and close-to-valid inputs can help test the core functionalities of the program, which should improve testing coverage and effectiveness as shown by previous techniques [13, 14].

Figure 1.1 shows code from a program (readELF) that accepts ELF files as an input. The code segment processes an ELF file's header, and it contains a bug that is located at a deeper location of the code segment. DASE can locate the bug because there is a branch condition at line 2 that checks if the magic number of the file is valid, which has to return false to reach the bug. DASE can infer the magic number by extracting constraints about the ELF file format's header from the documentation. DASE prunes away the execution path that contains an invalid file format, which helps the symbolic execution engine to reach the buggy location faster.

Challenges Since documentation are often written in natural language, it requires the utilization of natural language processing techniques and heuristics to extract the input constraints. The extraction of constraints from the documentation is a challenging process. First, English sentences may express the same information using a wide variety of terms and sentence structure. Therefore, we propose the usage of grammar rules to extract key information within each sentence. Second, source code comments and manual pages are semi-structured. They do not formally link the constraints back to the formally defined data structure, which requires static analysis and heuristics to link the data field constraints.

```
1  static int process_file_header(void) {
2    if (elf_header.e_ident[EI_MAG0] != ELFMAG0
3      || elf_header.e_ident[EI_MAG1] != ELFMAG1
4      || elf_header.e_ident[EI_MAG2] != ELFMAG2
5      || elf_header.e_ident[EI_MAG3] != ELFMAG3){
6        return 0;
7    }
8    ...
9    // An integer underflow bug in deeper location of the code
         .
10 }
```

Figure 1.1: Example on how DASE leverages constraints to discover bugs.

1.1.2 Automatic File Repair

We implement DocRepair to leverage documentation to help assist document recovery tools to repair corrupted files. Corrupted files cause many software failures. Specifically, file corruption causes software crashes [15], security issues [16, 17, 18, 19, 20], users unable to access valuable data, etc. For example, a corrupted PDF file triggered a null pointer bug in the Google Chromium browser, which caused the browser to crash [15]. Files often become corrupted due to problems such as file system bugs [21, 22], network transfer errors [23], malicious modifications of a file [24], and buggy software [25].

It is important to repair corrupted files to mitigate the resulting software failures. However, this task is challenging [26, 27]. File viewers often fail to display corrupted files because corrupted files do not follow PDF specifications and cannot always be parsed by the viewer. Thus, practical techniques to repair corrupted files are in high demand.

Challenges The extraction of file format constraints for document repair tools is challenging because it requires a good understanding of the corrupted file problem. It is important to understand the number of real-world corrupted files in the wild; the causes of file corruption (e.g., corrupted base structure and corrupted font); the impact of file corruption (e.g., incorrect functionality, crash, and performance degradation); and whether existing techniques can repair them. All these questions need to be investigated to design a solution for repairing corrupted PDF files.

There is neither a comprehensive study to quantify the existence of corrupted files in the real world nor a study to understand their impact, despite some anecdotal evidence [26] and previous work that analyzed the impact of data corruption in storage systems [28, 29, 30].

Therefore, we performed an empirical study to study corrupted files. Based on the results of the study, we implement a list of repair operations to repair the common causes of file corruption.

1.2 Automatic Documentation Generation

We propose and implement a new technique for automatic documentation generation. The purpose is to generate code comments automatically since code comments are scarce in practice. Code comment is a form of documentation that is written in natural language that contains information to help developers understand the source code. A better understanding of the source code helps developers to perform modifications to the target application and thus improves the software maintainability.

We implement CloCom+ to generate code comments by mining from two sources. CloCom+ mines code comments from existing software repositories in GitHub [6] and a Question and Answer site called StackOverflow [31].

CloCom+ has a similar design compared to our previously published work, AutoComment [32] and CloCom [7]. The main difference is that CloCom+ extracts code comments from both GitHub projects and Stack Overflow posts, instead of extracting comments from only one of the sources. CloCom+ adds a comparision against a previous work [33] with a user study, and performs an empirical study on the properties of automatically generated and human-written comments.

Figure 1.2 shows an example of automatic comment generation, where CloCom+ leverages a Stack Overflow post to generate comments automatically. The figure shows the title of the post, code snippet, and a paragraph that describes the code snippet from the answer. Since a similar piece of code segment also exists in the Java project—Vuze (not shown here), CloCom+ can extract the sentence from the post's answer, apply natural language processing techniques on the sentence, and generate following comment to explain the code segment in Vuze: *"Build this value in java using Calendar."*

The underlined term in the code segment represents the text similarity term between the sentence and the code segment, and we applied natural language processing techniques [34, 35] to refine the sentence into a source code comment.

Stack Overflow Question (Title):
How to manage dates in Android sqLite database when I care only about the date (which is to be unique) and not the time?
Stack Overflow Answer:
Alternatively, you can build this value in java using Calendar:

```
1 Calendar calendar = Calendar.getInstance();
2 calendar.set(Calendar.HOUR_OF_DAY, 0);
3 calendar.set(Calendar.MINUTE, 0);
4 calendar.set(Calendar.SECOND, 0);
5 calendar.set(Calendar.MILLISECOND, 0);
6 long time = calendar.getTimeInMillis();
```

Figure 1.2: Stack Overflow Post #32215979

1.2.1 Mining Question and Answer Sites

Stack Overflow contains code segments together with their descriptions, which we refer to as code-description mappings. We extract such mappings and leverage them to generate comments automatically for similar code segments matched in open source projects.

We mine source code comment from Q&A sites because they naturally contain code descriptions written by developers that can be used for automatic comment generation. For example, a user asked, "*Can I know if a given method exists?*" (Stack Overflow post `#28069121`). The question received a Java code snippet that checks if a given method had been declared in the class. We can use the statement form of the question "*Know if a given method exists.*" as an explanatory description of the code snippet.

Challenges It is a challenging task to select the correct sentence that describes the source code. First, a piece of code segment may have several sentences describing it. We have to select the sentence that best describes the code segment by leveraging text similarity heuristics. Second, the sentences that we mine from Stack Overflow cannot be directly utilized as a source code comment. We have to leverage natural language processing techniques to refine the sentence because a sentence can be in question form or contain unneeded information. Third, we have to detect for similar code segments between Stack Overflow and the input project. Since code segments in Stack Overflow are often not compilable. We apply refinements to the code segments before the application of the code clone detection process.

1.2.2 Mining Code Repositories

Existing software repositories contain source code comments that can be extracted for automatic comment generation. For example, previous work had shown that GitHub [6] contains 17 million lines of Java comments and 42 million lines of code across 1,005 projects (based on CLOC) [7]. Mining from existing code repositories brings a new set of unique challenges compared to the mining from Q&A sites.

Challenges Mining human-written comments from existing software repositories have four main challenges. First, since we mine code comments from a large pool of software repositories, a code segment is often similar to many other code segments that contain code comments because software reuse is common. Therefore, it requires a comment selection technique that works with a large set of candidates. Second, it is challenging to parse code comments from the source code. The reason is that code comments from existing software repositories are often not written in full sentences, which means a natural language parser cannot process the sentences accurately. Third, code comments mined from source code are more likely to contain project-specific information compared to human-written sentences on Q&A sites. Fourth, we require a highly scalable code clone detection tool [36, 37] that allows us to extract fine-grained code information such as type information and variable scope level. Therefore, we built a code clone detection tool that leverages an abstract syntax tree parser to improve the accuracy of the code clone detection technique.

1.3 Contributions

This thesis includes the following contributions:

- We propose and implement DASE for automated documentation analysis to automatically find bugs in the target software. DASE leverages documentation to help guide a symbolic execution engine to utilize constraints to focus the testing on execution paths that are semantically more important. We propose a new technique that combines natural language processing techniques and heuristics to extract the required constraints from manual pages and source code comments.

- We propose and implement DocRepair to leverage documentation to guide document recovery tools to repair corrupted files. We create the first dataset of real-world corrupted files and performed an empirical study to study their causes and impact on the end user and target application. Based on the result of the empirical study

we propose seven repair operators that utilize documentation constraints to repair corrupted files.

- We propose and implement CloCom+ for automatic documentation generation. CloCom+ generates source code comments through mining both existing software repositories from GitHub and a large scale Q&A site, Stack Overflow. It leverages natural language processing techniques to map sentences against code segments to generate a code-description mapping database for automatic comment generation, and it leverages information from an abstract syntax tree parser to perform text similarity for selecting the code comment that best describes the target code. Although CloCom+ improves on previous work, SumSlice, on automatic comment generation, the quality (evaluated on completeness, conciseness, expressiveness, and usefulness) and yield (number of generated comments) are still rather low which makes the technique far from ready for real-world usage.

- We propose future work to improve symbolic execution-based software testing on structured-file parsing applications such as XML and PDF, which requires a more expressive way (e.g., context-free grammar) to describe the file format. We describe a preliminary approach to extract file format constraints automatically from documentation using regular expressions and natural language processing techniques, where the extracted constraints can be applied on a file parser to help detect constraint violations on the target file.

1.4 Overview of Thesis

The following is an overview of the thesis. Chapter 2 discusses the related work and background information. We describe the two techniques for documentation analysis. Specifically, we utilize documentation to improve a symbolic execution tool (Chapter 3) and a document recovery tool (Chapter 4). Chapter 5 describes a technique that is used for documentation generation. Specifically, we generate source code comments by mining both GitHub projects and a Q&A site called Stack Overflow. Chapter 6 discusses the future work on automatically extracting documentation constraints using regular expressions and natural language processing techniques for bug detection. Chapter **??** provides an overview of my research publications.

Chapter 2

Related Work

This chapter introduces the background material and related work that are related to this thesis.

2.1 Symbolic Execution

Symbolic execution is a software testing technique that generates test cases automatically to expose bugs in a program. Instead of executing programs with concrete inputs, symbolic execution represents inputs as symbolic values. Upon exploring a branch whose condition involves symbolic values, two paths are created, and the corresponding constraints are added to each path. Once the execution of a path terminates, the collection of constraints along that execution path is used to generate concrete inputs to exercise the path.

Symbolic execution [38, 39] (alone or with concrete execution) has been widely used for automated testing [40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51]. However, it suffers from the fundamental problem of path explosion. To alleviate the path explosion problem, many strategies have been proposed [52, 40, 53, 46, 54, 55, 56, 57, 58, 59]. Veritesting [53] leverages static symbolic execution to guide and improve dynamic symbolic execution. CUTE [46] and CREST [55] both use a bounded depth-first search (DFS) strategy. ZESTI [60] uses developer generated tests as "seeds" and explore paths similar to the seeds' paths. ZESTI's performance is affected by existing tests. Differently, we propose DASE to utilize input constraints as a "filter" to aid search strategies to focus on both valid and close-to-valid inputs (e.g., boundary cases) to explore deeper in a program's core functionality.

DASE complements ZESTI: *DASE detected two previously unknown bugs in* COREUTILS *that were not detected by ZESTI (the same version of* COREUTILS *was used by DASE and ZESTI).* Input space partitioning [61] has been used to improve symbolic execution [62, 63, 64]. For example, `FlowTest` [62] partitions the inputs into "non-interfering" blocks by analyzing the dependency among inputs.

Recent work by Trabish et al. proposed chopped symbolic execution, which allows the user to manually specify uninteresting parts of the code (list of functions along with specific call sites) to skip to improve the performance of symbolic execution [65]. Their proposed technique, Chopper, lazily executes the code that user specified to skip. The evaluation is conducted on GNU libtasn1 which is a library that serializes and deserializes data in Abstract Syntax Notation One syntax. The authors manually created an execution driver for the library and manually specified a set of functions to skip. Our proposed technique, DASE, differs from Chopper in two ways in terms of the way it specifies the uninteresting parts of the code. First, DASE specify the interesting parts of the code by constraining the standard input of the program whereas Chopper specifies them through annotation of the source code. Second, DASE automatically extracts the constraints from the source code using natural language processing techniques and heuristics, whereas Chopper requires manual annotation on the functions.

One of the issues with symbolic execution is that it would fail to locate bugs that are within the deeper parts of the code when the bug requires a complex input to trigger. Concolic execution is a technique that is used to help symbolic execution explore deeper parts of a program by providing the symbolic execution engine with a concrete execution path. However, concolic execution still suffers from the path explosion issue due to the large number of execution paths in a program which limits the scalability. Previous work proposed hybrid fuzzing which combines fuzzing and concolic execution [66, 67].

Previous work by Stephens et al. proposed Driller [67] which combines concolic execution with fuzzing to reach deeper parts of the code by using fuzzing to alleviate path explosion. Driller utilizes the fuzzer to explore the initial compartment of the program (defined as checks for particular values of a specific input), and when the fuzzer is unable to identify inputs to reach new compartments of the program, it invokes the concolic execution component to determine the specific input that is needed to reach new paths. The new possible input that is determined by the concolic execution component is then once again passed back to the fuzzer to explore the input for the new component. DASE also attempts to infer the valid input from the documentation to reach new compartments of the program. Specifically, the inferred input from DASE is utilized to reach beyond the compartment that process the command line options and the compartment that processes the program's input file. One of the main advantages of Driller over DASE is that

Driller does not necessarily require an input test case to operate, where DASE requires the standard input of the program to be constrained with file format constraints. Driller is also capable of obtaining new constraints that are required to explore deeper parts of the program dynamically during concolic execution as more code is covered, where DASE determines all the required constraints prior to the execution of the symbolic execution tool.

Previous work by Yun et al. proposed QSYM [66] which also combines fuzzing with concolic execution. The main focus of QSYM contains several performance improvements and optimizations. QSYM contains optimizations including the removal of the IR translation layer from the symbolic emulation, removal of the snapshot mechanism from hybrid fuzzing, and the collection of an incomplete set of constraints for efficiency. The optimizations enabled QSYM to become much more scalable. The evaluation of QSYM shows that it is able to outperform Driller in terms of code coverage for 104 out of the 126 DARPA CGC binaries.

The previous techniques rely on information from the code logic to guide the path exploration process. Differently, DASE automatically extracts input constraints from documents and uses the constraints to prune execution paths. Also, DASE focuses on valid and close-to-valid inputs while the above techniques have no knowledge about whether an execution path corresponds to valid or invalid input.

Symbolic execution by default is an exhaustive testing approach that attempts to explore the entire input space on the target software, which is the reason that it is not scalable. Previous work proposed Directed random testing [68], where it combines random input generation with a heuristic pruning technique that discards illegal and equivalent inputs to reduce the input space. Directed random testing is more scalable compared to symbolic execution, but it also lacks a meaningful stopping criterion since it does not systematically explore the program.

2.2 Automatic File Repair

Much work focused on automatic file repair. The first branch of work relies on manually written constraints. Demsky et al. detect and repair violations on data structures using manually-written constraints [69], and Endignoux et al. proposed manually written formal grammar rules to validate the PDF file format [70]. The second branch of work repairs the corrupted file without knowledge of the file format. Docovery [26] uses symbolic execution to modify potentially corrupted bytes of a file and avoid program paths that will lead to

a crash. SOAP [71, 72] automatically learns the constraints of a file format from training inputs, but the extracted constraints are based on the patterns observed in the input examples and can potentially be incorrect. The third branch of work repairs a corrupted file by modifying the execution path of the file viewer, which does not modify the corrupted file. Wüsta et al. proposed Force Open [27] to repair corrupted files with a black box approach using binary instrumentation. The fourth branch of work is the specialized repair tools. In the area of the PDF file format, it includes repair tools (e.g., MuPDF [73], PDFtk [74], PDF Repair Tool [75] and PDF Repair Toolbox [76]) and parsing libraries (e.g., Poppler [77] and GhostScript [78]). These tools are developed by experts to parse and repair specific file formats. In this work, we propose DocRepair, as a specialized repair tool since we rely on expert knowledge from the study.

Rinard [72] proposed a manually defined rectifier to transform program input into a constrained input format for email messages. The purpose is to avoid unanticipated errors and vulnerabilities during the program execution. A follow-up work, SOAP [71], automatically infers constraints based on a set of training inputs. SOAP learns a set of constraints such as the upper bound and constraints of integer fields (i.e., sign of the value). However, the learned constraints are dependent on the training set of the program, which can be incorrect if all the input files have the same value in specific fields.

Kuchta et al. proposed Docovery [26], a document recovery technique based on symbolic execution that is independent of the input file format. The technique first identifies the input bytes that cause the program to crash using dynamic taint tracking. Then, it executes the program concolically [79], and upon reaching a point where the program would crash, it forces the program to follow a different execution path by modifying only the bytes in the document that are related to the crash. The technique works well for less complicated document formats. However, PDF proved to be a challenging target due to the complexity of the file format.

Docovery technically can theoretically repair any corrupted given an unlimited amount of time to allow the symbolic execution engine to reach the exit of the program without any errors. However, existing symbolic execution tools still have scalability issues beyond small applications. On the other hand, DocRepair's repair capability is limited towards the defined list of repair operators. Docovery generates a large number of repair candidates (that may or may not be repaired) since it creates a candidate file for each execution path that leads to a crash, which differs from traditional repair tools (e.g., DocRepair, Mutool, PDFtk, and GhostScript) that generates a single repaired file. Since Docovery generates multiple repair candidates, it would require a tool to select the best candidate from the list of files.

Wüsta et al. proposed Force Open [27] to automatically repair file formats including PNG, JPG, and PDF files with a black box approach that relies on binary instrumentation. Unlike existing approaches that repair a file by modifying bytes of the corrupted file, it modifies the execution of the file viewer to force it to open a corrupted file. Force Open learns the behavior of a file viewer on valid files and modifies the program to follow learned behavior to force open a corrupted file. They compared their repair capability against a PDF repair tool called PDFtk [74], and their technique achieved a repair rate of 4.9% compared to PDFtk's 13.4%.

Rinard et al. proposed a technique to detect and repair errors in data structures for several applications (e.g., a Linux file system and Microsoft Office files) [80]. It allows manual specification of the data structure's constraints, which are represented by a set of object and relation declarations.

Brown et al. proposed the reconstruction of corrupt files that utilize the DEFLATE compression algorithm (e.g., ZIP archives, Microsoft Office 2007 and OpenDocument documents) [81, 82].

2.3 Automatic Comment Generation

Code commenting is an integral part of software development. It helps improve software maintainability [83] and programming productivity by helping developers understand the source code.

Several automatic comment generation techniques generate source code comments automatically for certain code structures, such as failed test cases [84], the context surrounding a method [33], exceptions [85], APIs [86], code changes [87] and function parameters [88]. Sridhara et al. proposed an approach to generate comments automatically from source code for Java methods [89], high-level actions within methods [90], and Java classes [91]. These techniques rely on the source code to contain high-quality identifier name and method signatures to generate a comment. For example, when grouping method calls [90] such as `buildGameMenu` and `buildViewMenu`, all the method names have to contain the same verb, `build`. It would then synthesize `build menus` or `build different menus` as the output. These techniques [89, 90, 91] all leverage the Software Word Usage Model [92], which is responsible for capturing the action, theme, and secondary arguments for a method. Their technique [90] works on statement sequences that are conditional blocks, perform similar actions, or follow specific templates. Differently, we propose AutoComment and CloCom, which can generate a high-level comment for multiple statements that perform different actions, and it is not limited to the grouping strategy.

Previous work generates comments for software concerns [93] and MPI programs [94]. These techniques do not solve the problem of grouping statements that perform *different* sub-actions into a high-level action. Recently, Wang et al. [95] proposed a grouping strategy that segments method code into meaningful blocks. The grouping strategy can potentially improve the previous technique [90], but it is still difficult to generate comments for a group of statements with *different* sub-actions. Differently, AutoComment can naturally group the statements based on the code segments from Stack Overflow written by developers, and CloCom does not require grouping of the statements.

Recent work by Hu et al. proposed a technique called DeepCom to generate code comments for Java methods automatically by analyzing the structural information of the code, which utilizes machine translation techniques to convert source code to natural language sentences [96]. DeepCom treats comment generation as a Neural Machine Translation (NMT) problem, and it builds a language model for both the source code and comments. They proposed a unique way to translate the abstract syntax tree (AST) of the code to a specially formatted sequence to ensure the code representation is lossless for the language model. The evaluation of their work is compared against an existing technique called CODE-NN [97] which utilizes recurrent neural network (RNN) with attention to generate summarizes, and they utilize the BLEU score to measure the performance. Our proposed technique, CloCom+, generates code comment for code segments within a Java method, as opposed to DeepCom which generates code comment for Java method.

The evaluation of code comments and code summaries commonly involves recruiting multiple human participants to judge the generated comment and summary. Moreno et al. evaluated Java class summaries using criteria including content adequacy, conciseness, and expressiveness [91], and Sridhara et al. evaluated Java method summaries using accuracy, content adequacy and conciseness [89]. However, these criteria all focus on the intrinsic evaluation instead of the extrinsic evaluation. An intrinsic evaluation focuses on evaluating the content of the summary (e.g., accuracy, adequacy and conciseness), whereas an extrinsic evaluation focuses on the usefulness of the summary (e.g., impact on the developers). For example, an extrinsic evaluation can measure how much faster a developer can perform a coding task while given additional resources such as code summaries.

Figure 2.1 shows a code segment from the Java project—Freecol. Previous work from [90] generates comments for high-level actions within methods. Their work generates comments for three types of statement groups. The first type involves sequences of statements that can be grouped into a high-level action (same verb phrase with a common headword). The second type involves conditional blocks that contain integrable sequences of statements along the branches (`if ... else if ... else ...` or `switch`). The third type involves specific common high-level code patterns that are based on loop constructs. Their

```
1  while(wNew*2 <= w && hNew*2 <= h) {
2    w = (w+1)/2;
3    h = (h+1)/2;
4    BufferedImage halved = new BufferedImage(w, h,
       BufferedImage.TYPE_INT_ARGB);
5    Graphics2D g = halved.createGraphics();
6    // For halving bilinear should most correctly average 2x2
       pixels.
7    g.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
       RenderingHints.VALUE_INTERPOLATION_BILINEAR);
8    g.drawImage(im, 0, 0, w, h, null);
9    g.dispose();
10 }
```

Figure 2.1: Code from Java project—Freecol

work cannot generate a comment for this code segment. The code contains no sequences of statements that can be grouped together. For example, method `setRenderingHint()` and `drawImage()` contain different actions, set and draw. The code also does not contain any conditionals or loop constructs.

Previous work from [89] generates summaries for Java methods, which are leading comments that occur before a method. For example, it can generate a summary for the method, `scaleImageTo`, under Figure 5.2. Their technique first selects important lines of code that are important to a method summary. It selects the return statement on line 12 because it is the exiting point of a method; selects line 7, 10 and 11 because they contain method calls that do not have a return value; and selects line 6 because the return value, `g`, is not in the return statement. It then performs filtering and removes line 6 because the method, `createGraphics`, contains an object creation operation. In the end, it synthesizes natural language sentences for line 7, 10, 11 and 12 using their Software Word Usage Model, which then becomes the method summary. Our work focuses on synthesizing comments for high-level actions within methods instead of synthesizing comments for the entire method.

2.4 Source Code Summarization

Source code summarization presents the set of important keywords that best represents the source code. It differs from automatic comment generation, where comment generation requires the generated sentences to describe the functionality of the source code.

Previous work utilizes text retrieval (TR) techniques (e.g., Vector Space Model, Latent Semantic Indexing and hierarchical PAM) to summarize source code [98, 99, 100, 101]. McBurney et al. [102] proposed a technique that utilizes topic modeling technique to select keywords to represent the source code. The approach is similar to the traditional field of automated text summarization. A source code summary contains terms that are extracted from the identifiers and comments in the source code. TR technique selects terms based on the weight of each term in the source code document, and the terms with the highest similarity against the document are captured in the summary. These source code summarization techniques do not generate natural language sentence summaries, where they choose the top K terms to represent the source code.

Moreno et al. [103] performed an analysis of human-written and automatic summaries. The result shows that developers create longer natural language summaries of code sequences that involve sequences of method calls, and that term-based summary is less informative compared to a sentence-based summary. They also showed the key structural elements that should be included in a summary. This thesis proposed CloCom+ which specifically generates sentence-based summaries.

2.5 Mining Descriptions for Code Artifact

Many studies mine descriptions or documentation for code artifacts from developers' communications, such as bug reports [104], forum posts [105], issue trackers [106] and the web [107]. These studies focus on project-specific descriptions. For example, they extract descriptions for Eclipse code artifacts from the mailing list and bug tracking system of Eclipse. Thus, such descriptions are more likely to benefit Eclipse.

Differ from previous work we propose AutoComment to mine descriptions that are general for each domain. AutoComment extracts *source code comments* that directly describes the functionality of the code segment, whereas previous work mines information that is specific to the project. AutoComment is not limited to generating descriptions for methods, and we adapt NLP techniques to improve the comment quality. Also, previous work leverages heuristics (e.g., text similarity) to link descriptions and code. AutoComment combines clone code detection and heuristics to improve accuracy. Some work helps improve code extraction from unstructured data, such as emails and documents [108, 109]. In the future, AutoComment can leverage these techniques to mine more descriptions from emails and documents, not only from Q&A sites.

Recent previous work by Chatterjee et al. [110] performed an exploratory study to learn the different kinds of information (e.g., testing, efficiency, erroneous, design, etc.)

that are embedded in different types of software documents (e.g., Q&A sites, bug reports, documentation, etc.). Chatterjee et al. [111] also proposed a technique to automatically identify code segments, along with the corresponding descriptions, from research articles.

2.6 Code Clone Detection

There are three major types of code clone detection techniques: token-based [112, 113, 114], AST-based [115, 116], and semantics-based [117]. Roy et al. performed a comprehensive comparison and evaluation of the current state-of-the-art in clone detection tools [118]. In our proposed work, AutoComment and CloCom, we developed a token-based code clone detection tool to address both the scalability and adaptability of the code clone detection process. It supports the compilation of partial code segments in Stack Overflow. Its clone matching algorithm provides better support for detecting code clones that contain inserted and deleted statements.

CloCom [7] utilized a code clone detection tool with a matching algorithm similar to that of existing work, DuDe [119]. CloCom utilized an abstract syntax tree parser for tokenization of the code elements because the source code can be compiled, where SIM [112] used a custom tokenizer written in Lex. CloCom's code clone detection tool is more scalable compared to SIM [112], and its clone matching algorithm provides better support for detecting code clones that contain inserted and deleted statements.

Previous work from Wang et al. proposed a token-based large-gap clone detector called CCAligner [120]. CCFinderX [114]. Sajnani et al. proposed a token-based clone detector called SourcererCC that utilizes a similarity overlap threshold to detect type-3 clones [37]. Svajlenko et al. addressed the scalability issue of code clone detection tools. They developed a shuffling framework [121] to scale classical code clone detection tools to ultra-large datasets for tools such as Deckard [115], NiCad [122], iClones [123] and CCFinderX [114].

2.7 Fuzz Testing

Fuzz testing is a software testing technique that inserts random, invalid or unexpected data into a program to locate bugs [13, 124, 125]. Blackbox fuzzing provides a program with either random inputs or invalid inputs that had been mutated from a well-formed input. Whitebox fuzzing [13] utilizes symbolic execution to collect constraints on the program input, which allows mutation of the input such that it impacts the execution path of the program.

Godefroid et al. proposed SAGE [124] as a whitebox fuzzing technique for security testing. It utilizes symbolic execution to gather constraints of the program input to help find defects in structured-file parsing applications. It first runs the program in concrete with a well-formed input, which allows the collection of branch constraints on the individual input bytes. SAGE then negates the constraints one at a time to yield new inputs that can exercise different parts of a program.

Majumdar et al. proposed CESE [126] that combines grammar-based blackbox fuzz testing with symbolic execution. CESE relies on converting context-free grammar of the program input into symbolic grammar, enumerating a set of valid symbolic strings from the symbolic grammar, and performs concolic symbolic execution on the symbolic strings, where unbounded parts such as variable names and numbers are replaced with a symbolic constant.

Godefroid et al. extended SAGE and proposed a grammar-based whitebox fuzzing technique [13] that enhances whitebox fuzzing with a grammar specification of the valid input. The grammar specification is translated into higher-level symbolic constraints, which restricts the symbolic tokens that are returned by the lexer to valid inputs. Their approach differs from traditional techniques which marks input bytes as symbolic [127, 43, 124]. Instead, they constraint individual tokens as symbolic, which are marked by a lexer.

However, the above techniques require input grammar to be given manually, Therefore, we propose future work to extract context-free grammar that describes the program input automatically. A recent work by Lemieux et al. [128] proposed to automatically generate inputs with feedback-directed mutational fuzzing, which does not require any domain knowledge about the program.

2.8 Documentation Analysis

Many techniques analyze documents such as manual pages to check for undocumented error codes [129], and code comments [8, 130, 131] and API documentation [132] for bug detection. Much work focused on analyzing software documentation using NLP techniques. Our previous work, DASE [133], extracts constraints from documentation to guide symbolic execution to focus testing execution paths that implement program's core functionalities. The valid input of a program, which can be expressed using a set of input constraints, reduces the size of the search space of execution paths. This thesis attempts to utilize constraints from the documentation to repair corrupted files and detect bugs in programs.

Zhong et al. proposed work to inference resource specifications from API documentation

for bug detection [134]. The work extracts the action that the method takes and the resource that the method interacts against. Zhong et al. also have a follow up work that infers method specifications from API descriptions [135]. The former work utilized Hidden Markov Model for building the action-resource pairs, where the ladder utilized semantic templates.

Rubio-González et al. proposed work to detect inaccurate documentation by comparing Linux system calls against their respective manual pages [136]. Tan et al. utilized NLP techniques to detect inconsistencies between source code and code comments to detect software bugs [8, 130].

Chapter 3

Documentation Analysis: Symbolic Execution-Based Software Testing using Documentation Constraints

3.1 Motivation

Software testing is an essential part of software development. Many automated test generation techniques are proposed and used to improve testing effectiveness and efficiency [52, 137, 138, 40, 124, 139].

Symbolic execution [38, 39] has been leveraged to automatically generate high code coverage test suites to detect bugs [53, 140, 141, 142, 143, 60, 144]. Symbolic execution represents inputs as symbolic values instead of concrete values. Upon exploring a branch whose condition involves symbolic values, two paths are created, and the corresponding constraints are added to each path. Once the execution of a path terminates, the collection of constraints along that execution path is used to generate concrete inputs to exercise the path. Symbolic execution suffers from the fundamental problem of path explosion. In practice, one needs to use search heuristics and other techniques to guide symbolic execution [52, 55, 124, 62, 60].

Although symbolic execution has been successful in improving testing effectiveness, existing techniques do not take full advantage of programs' input constraints expressed in documents. Valid program inputs typically need to follow certain constraints. For example, `rm` (version 6.10) only accepts 11 options including `-r` and `-f`, and `readelf` requires its

input files to follow Executable and Linkable Format (ELF). Focusing on the valid and close-to-valid inputs can help test the core functionalities of the program, which should improve testing coverage and effectiveness as shown by previous techniques [145, 126]. It allows symbolic execution to devote more resources on testing code that implements program's core functionalities, as opposed to code for input sanity check and error handling. Fortunately, information about input constraints commonly exists in software documents, such as programs' manual pages (e.g., the output of `man rm`) and the comments of header files (e.g., `elf.h`).

Thus, we propose a general approach, *Document-Assisted Symbolic Execution* (*DASE*), to enhance the effectiveness of symbolic execution for automatic test generation and bug detection. DASE *automatically* extracts input constraints from documents, and uses these constraints as a "filter" to favor execution paths that execute the core functionalities of the program. DASE, as a path pruning strategy, can be used on top of existing search strategies to further improve symbolic execution (Section 3.6 shows that DASE can find more bugs and improve testing coverage on top of different search strategies).

This automation is novel because existing symbolic execution techniques [145, 126] do not analyze documents automatically and require input constraints to be given. Previous work has shown that constraint extraction from documentation [146, 147] is important yet challenging. Since this automation can reduce manual effort, DASE could make it easier for practitioners to adopt these symbolic execution techniques [145, 126] and other techniques that require input constraints [138, 148, 149] such as constraint verification.

DASE considers two categories of input constraints: the format of an input file (e.g., ELF and tar), and valid values of a command-line option (e.g., `-r` for `rm`). These two types are sufficient for a wide spectrum of programs. This work makes the following contributions:

- We propose a novel approach, *DASE*, to improve automated test generation. By leveraging input constraints automatically extracted from documents, DASE enables symbolic execution to automatically distinguish the *semantic* importance of different execution paths to focus on programs' core functionalities to find more bugs and test more code.

- We propose a new technique that combines natural language processing (NLP) techniques, i.e., grammar relationships and heuristics, to automatically extract input constraints from documents. The technique is general and should be able to extract input constraints for purposes other than symbolic execution such as program

comprehension and constraint verification. We study two types of documents, i.e., manual pages and code comments, and extract input constraints from both.

- Our evaluation shows that DASE finds more bugs and has higher code coverage than KLEE [40] (a symbolic execution tool without input constraints from documents). We evaluated DASE on 88 programs from 5 widely-used software suites—GNU COREUTILS, GNU FINDUTILS, GNU GREP, GNU BINUTILS, and ELFTOOLCHAIN, most of which have been thoroughly tested by many symbolic execution tools [40, 53, 60, 41]. DASE detected 12 previously unknown bugs[1] that KLEE failed to detect, 6 of which have already been confirmed by the developers, and KLEE only detected 2 previously unknown bugs that DASE failed to detect. Compared to KLEE, DASE increases line coverage, branch coverage, and call coverage by 14.2–120.3%, 2.3–167.7%, and 16.9–135.2% respectively, which are 6.0–21.1 percentage points (pp), 1.6–18.9 pp, and 2.8–20.1 pp increases. The input constraint extraction of three files formats—ELF, tar, and the Common Object File Format (COFF)—has accuracies of 97.8–100%.

3.2 Overview

A real-world program typically contains numerous or even infinite number of execution paths. Given limited time, it is crucial for testing to prioritize the paths effectively. Researchers have proposed approaches to guide the path exploration of symbolic execution [52, 55, 124, 62, 60] to find more bugs and improve code coverage.

Path pruning, which applies a "filter" to prune "uninteresting" paths before employing a search strategy, can further address the path explosion problem. Path pruning significantly reduces the size of the search space for a search strategy, which allows the symbolic execution engine to devote more time to test the "interesting" paths that would not have been tested due to time constraints.

We propose using *input constraint*s as a "filter" to aid search strategies to focus on both valid and close-to-valid inputs (e.g., boundary cases) to explore deeper in a program's core functionality. The core functionality of a program is typically related to processing valid inputs. For example, a C compiler's core functionality is parsing and compiling valid C programs. Valid C programs are only a small portion of all strings (the input space of a C compiler).

[1] We do not count bugs that are already reported in the KLEE paper. Those bugs, which can also be found by DASE, are not counted as newly detected ones by DASE either.

```
1  int counter = 0;
2  for (int i = 0; i < 30; i++) {
3     if (input[i] == 'A') {
4        counter++;
5        foo();
6     }
7  }
8  if (counter == 30) {
9     process_boundary_cases(); // bug!
10    if (input[30] == 'B')
11       process_valid_input(); // bug!
12 }
```

Figure 3.1: Motivating example on how input constraints help symbolic execution find more bugs.

Randomly generated inputs can cover many invalid inputs, but miss valid and close-to-valid ones. While *symbolic execution* addresses this issue by exploring paths systematically, it *is unaware of which branch (the "then" branch or the "else" branch) leads to valid inputs upon a conditional statement.* Input constraints that define valid inputs can guide symbolic execution to focus on paths corresponding to valid inputs. The constraints can be slightly relaxed (e.g., relaxing a constraint "x must be between 0 to 10 (inclusive)" to "x must be between -1 and 10 (inclusive)") to exercise paths corresponding to close-to-valid inputs to test boundary cases.

These paths (for valid and close-to-valid inputs) can pass the trivial part of input sanity check to go deeper and are more likely to uncover bugs [145, 126] for two main reasons. First, keeping invalid inputs in the search space hurts the effectiveness of symbolic execution based test generation. The reason is that exploring invalid inputs takes up time and memory, which can be used for testing valid and close-to-valid inputs instead. Second, some constraints are solved or simplified (e.g., the ones related to the concrete valid option), which reduces the computation time of the constraint solver.

Next we (1) illustrate why input constraints can help symbolic execution find more bugs and improve testing coverage and (2) summarize how DASE extracts these types of constraints automatically from two sources of documents.

Why can input constraints help symbolic execution find more bugs and test more code? We will use the code snippet in Figure 3.1 to answer this question, while real

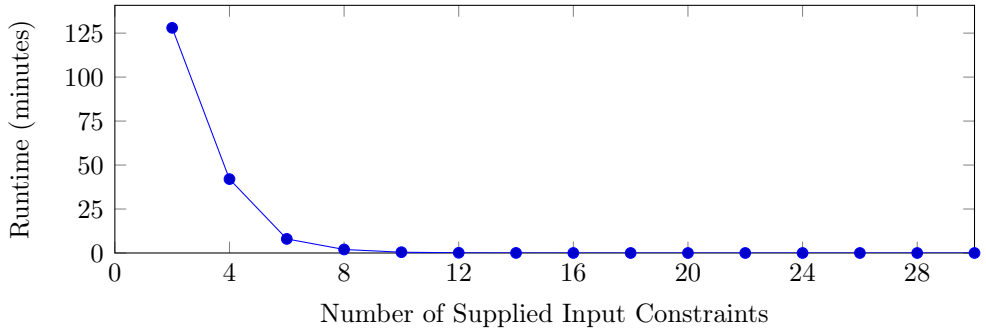
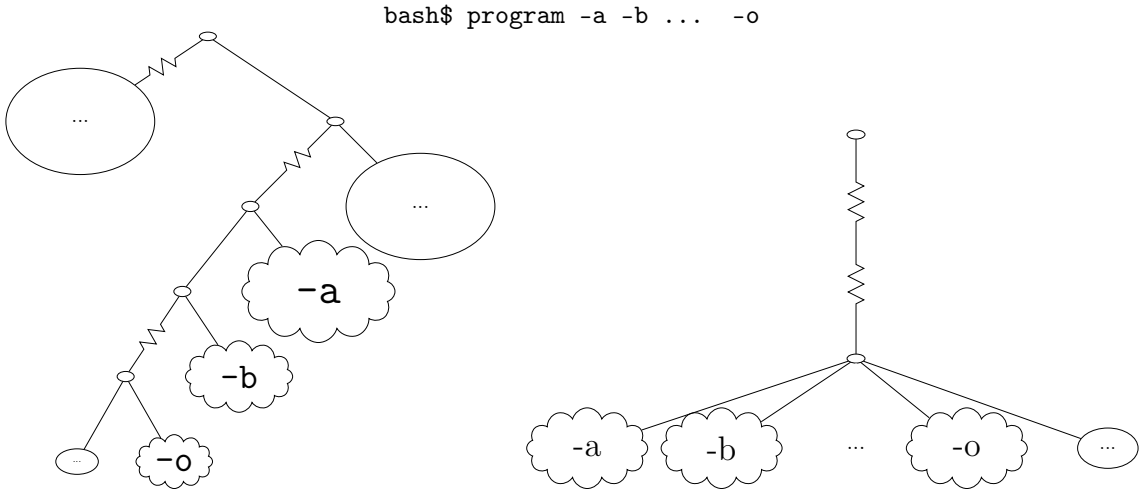


Figure 3.2: The runtime to find the bug in line 11 decreases exponentially as we supply more input constraints. The runtime when no constraint is supplied is not depicted because the bug was not detected after 10 hours when we stopped the execution.

code from BINUTILS is shown later in Figure 3.5 to explain how the automatically extracted input constraints help DASE detect previously unknown bugs and improve coverage. The code snippet in Figure 3.1 has 32 branches (30 from line 2 and 3, one from lines 8 and one from line 10), indicating 2^{32} possible paths to explore. Without knowing which paths execute the core logic, it is hard to expose the bug deep in line 11 because only 1 out of the 2^{32} paths leads to that line. DASE automatically extracts constraints from documents and find that the first 30 characters of a valid input must be 'A', and the next character must be 'B'. These constraints will guide the execution to line 11. If the document is incomplete, e.g., only mentioning that the first 30 characters of a valid input must be 'A', we can still hit the bug in line 9 that is triggered by close-to-valid inputs. In addition, it increases the chance to detect the bug in line 11 ($\frac{1}{2^{32}}$ to $\frac{1}{2}$). In either case, DASE can cover more code (lines 9–10 and possibly 11), which is hard for standard symbolic execution to cover, in addition to detecting more bugs.

We run KLEE on this example for 10 hours, and KLEE detects neither of the bugs. In contrast, DASE detects both bugs in 0.1 seconds. In practice, one may not have all constraints to define the entire input. In order to understand the effect of the number of constraints, we plot how the time to discover the bug in line 11 changes as the number of given constraints changes in Figure 3.2. The runtime to find the bug decreases exponentially as we supply more input constraints, suggesting that input constraints can dramatically improve the efficiency of finding bugs, i.e., finding more bugs given the same amount of time.

How to flatten symbolic execution to find more bugs and test more code? Command-line options are a special type of input. Therefore, we propose a new way to leverage their constraints to improve testing effectiveness. Command-line options are



(a) Without DASE (b) With DASE

Figure 3.3: Abstract view of execution trees for command-line options. Clouds are execution subtrees related to valid command-line options. Ovals are other execution subtrees. Deep options such as `-o` are more likely to be tested with DASE.

used to invoke certain functionalities of programs or tune parameters. For example, the option `-r` tells `rm` to perform a recursive deletion. A program typically uses nested if-else statements or a switch-case statement to check the input argument against all valid options until it finds a match, and then invokes the corresponding functionality.

DASE extracts valid options by analyzing programs' documentation and use them as input constraints. For example, DASE finds that among the 256^m possibilities[2] for `rm` (m is the maximum number of characters allowed in an option), only 11 values are valid options. With n valid options ($n = 11$ in the example above), DASE "forks" the execution state n times, with each child execution state taking a valid option[3]. In this way, DASE creates n execution branches for a program with n valid options (one for each valid option).

The concretization moves all valid options at the same depth of the execution tree, indicating that all valid options are treated equally (Figure 3.3b). Figure 3.3a illustrates the dynamic execution tree of symbolic execution without DASE. Clouds are subtrees related to valid command-line options. If a program has 15 valid options `a`–`o`, and `o` is the deepest valid option as shown in Figure 3.3a, the time spent on testing code related to

[2]There are 256 possibilities for a single 8-bit character option.

[3]We have additional child execution states for an invalid option and a null option for completeness. But the execution time for these two options should be less than the circles in Figure 3.3a combined.

option `-o` could be $\frac{1}{2^{15}}$ of the total testing time without DASE[4]. The reason is there are 15 branches from the nested if–else or switch-case statements, one for each valid option. Such a small fraction often means option `-o` would not be tested at all in practice. With DASE, the time of testing option `-o` would be much longer—about $\frac{1}{15}$ as in Figure 3.3b. This way, a bug in the code that processes option `-o` will be more likely to be exposed with DASE. On the other hand, the probability of hitting a bug in a shallower option (e.g., `b`) would be reduced from $\frac{1}{2^2}$ to $\frac{1}{15}$, but the difference is much smaller, and it is still highly likely that the option `b` will be tested given that the probability is $\frac{1}{15}$. In addition to finding more bugs, since each option has about $\frac{1}{15}$ chance to be explored, more options are likely to be tested, improving testing coverage (Section 3.6.2 shows that DASE covers more options than KLEE).

Although this approach may appear to be similar to breadth-first search (BFS), it is very different from BFS. Without DASE, BFS would explore paths in Figure 3.3a, which would still waste time on shallow paths and are less likely to explore deeper paths. In fact, our evaluation shows that DASE outperforms KLEE even if BFS is used as the underlying search strategy (Section 3.6.2).

What documents to analyze and how to extract input constraints from them automatically? Many types of software documents are available: manual pages, code comments, API documentation, requirement documents, etc. This work studies and analyzes two popular types for constraint extraction, i.e., manual pages and code comments, since they describe whole program constraints (as opposed to API documentation that describes method level constrains), and they contain more code-level constraints (compared to requirement documents). We conduct an informal qualitative study of 82 manual pages from Coreutils and code comments of 3 header files (ELF, tar and COFF). **Manual pages** typically have higher English quality (e.g., grammatically correct full English sentences) since they are meant to be read by more than just the developers. On the other hand, it is easier to link constraints from **code comments** to code artifacts since comments are embedded in the code (e.g., a comment typically describes the code segment right below it).

Valid options are typically described in a well-structured manner in manual pages. Therefore, we use simple regular expression matching to extract them (Section 3.4.3). Input file formats are described in both manual pages and code comments. We use regular expression matching to analyze the manual pages. Since code comments are less structured, we use NLP techniques, i.e., grammar relationships, for extraction (Section 3.4.1). Gram-

[4]The actual time depends on the search strategy, but the time spent on testing `-o` would be much smaller than that of testing `-a`.

mar relationships can help identify relevant sentence structures for constraint extraction. It can tolerate different word orders and paraphrases, thus more general than hard-coded heuristics.

3.3 Background

KLEE is a symbolic execution engine based on LLVM. Programs are compiled into LLVM bytecode, and then interpreted by KLEE. KLEE models the programs' running states. It checks for dangerous operations (e.g., pointer dereferences and assertions) that can cause the program to fail. In addition, KLEE maintains path constraints that drive the execution to the current state. KLEE provides a function `klee_make_symbolic()` to make the memory symbolic, whose usages are tracked and constraints are collected. KLEE can also intercept the startup of programs and insert logic to make them support options for symbolic execution by using function `klee_init_env()`. Supported options include (1) `--sym-args MIN MAX N`, which expands to at least `MIN` and at most `MAX` symbolic arguments, each with a maximum length of `N`; and (2) `--sym-files NUM N`, which makes `stdin` and up to `NUM` files symbolic, each with a maximum size of `N`.

KLEE's default search strategy consists of two atom search strategies that are interleaved in round-robin fashion to prevent one atom strategy from getting stuck. The first atom strategy, coverage-optimized search, uses heuristics to choose a state that is most likely to cover new code in the immediate future. The second atom strategy, random path selection, randomly selects a branch to follow at a branch point, which helps alleviate starvation.

3.4 Design and Implementation

This section describes how DASE extracts and utilizes input constraints for file formats (Section 3.4.1 and Section 3.4.2) and options (Section 3.4.3 and Section 3.4.4).

3.4.1 Extracting File Format Constraints

DASE automatically extracts input constraints regarding file formats from both code comments and manual pages. As discussed in Section 3.2, code comments and manual pages have different characteristics, so different techniques are used to extract constraints from

them: NLP techniques for code comments, and regular expressions for manual pages. The same techniques are used for all three file formats—ELF, tar, and COFF.

We apply NLP techniques to analyze the comments and code in header files to extract constraints automatically. The header file contains a large number of comments that describe the constraints for the struct data fields (i.e., each comment is followed by a list of macros representing the valid values). One example is:

```
/* Fields in the e_ident array. The EI_* macros are
   indices into the array. The macros under each
   EI_* macro are the values the byte may have. */
#define EI_MAG0     0
#define ELFMAG0     0x7f
#define EI_MAG1     1
#define ELFMAG1     'E'
```

DASE automatically generates two constraints regarding array index-value pairs from the comments and code:

```
assume(Elf32_Ehdr->e_ident[EI_MAG0] == ELFMAG0);
assume(Elf32_Ehdr->e_ident[EI_MAG1] == ELFMAG1);
```

where `assume()` is a KLEE function for putting constraints onto the current path. The rest of this section explains the NLP techniques to generate the constraints.

Our technique extracts two types of value constraints: array index-value pairs and struct field values (e.g., `assume( Elf32_Shdr->e_type == 0|...);`). Since comments are written in natural language, developers can use different forms to express the same meaning. For example, they may use "Fields in the e_ident array", "Fields of the e_ident array", "The e_ident array's fields", or "The array e_ident's fields" to start the listing of fields. These sentences use different sentence structures and words to express the same meaning, which are difficult to analyze automatically. Simple regular expression matching will fail to accommodate all these and other variants.

We propose to use Stanford typed dependency [150] to analyze the dependencies and grammatical relations among words and phrases in a sentence to handle these variants. Our technique is different from prior work [8, 151].

DASE uses four grammar rules (GR) to identify relevant comments and extract constraints from them. All four rules are used as main rules to identify relevant comments—if a sentence contains the typed dependency defined by a GR, it is considered relevant and

remains for further analysis. GR1 and GR2 can also act as a supporting rule for any main rule. For example, GR1 can help identify the parameters in a rule, e.g., array and field names. The four GRs are listed below:

- **GR1: Noun or Adjectival Modifier (main/support rule)** Noun or Adjectival modifier is a noun or adjectival phrase that modifies a noun phrase [152]. For example, in the comment "Fields in the e_ident array", the noun phrase "e_ident" modifies the noun "array". DASE applies this grammar relationship to retrieve data structure names and index names.
- **GR2: Prepositional Modifier (main/support rule)** Prepositional Modifier is a prepositional phrase that modifies the meaning of a verb, adjective, noun or preposition [152]. For example, in the comment "Legal values for sh_type field of Elf32_Shdr", the prepositional phrase "for ... Elf32_Shdr" modifies the noun "values". DASE applies this grammar on modifiers (i.e., "for", "of", "in" and "under") to locate specific nouns (i.e., "value" and "field") or specific word in the prepositional phrase (i.e., "field").

  After locating the prepositional modifier the dependency tree links "values" to the content word "field". If the content word is being modified by an adjectival modifier, DASE applies GR1 to resolve the properties. In this example, GR1 will return "sh_type" as the property of "field", and GR2 will flag the macros as the legal values for that data field.
- **GR3: Nominal subject (main rule)** Nominal subject is a noun phrase that is the syntactic subject of a clause [152]. For example, in the comment "The EI_* macros are indices into the array". The noun, "macros", is the subject of the clause, "indices into the array". DASE applies this grammar to locate specific clauses (i.e., "indices ..." and "values ...").

  After locating the nominal subject, DASE applies GR1 to resolve the properties. In this example, GR1 will return the regular expression "EI_*" as the property of "macros", and GR3 will flag the macros named under this regular expression as the indices of an array.
- **GR4: Possession modifier (main rule)** Possession modifier holds the relation between the head of a noun phrase and its possessive determiner [152]. For example, in the comment "sh_type field's legal values". The head noun is "field" and the possessive determiner is "values". DASE applies this grammar to locate specific possessive determiners (i.e., "value").

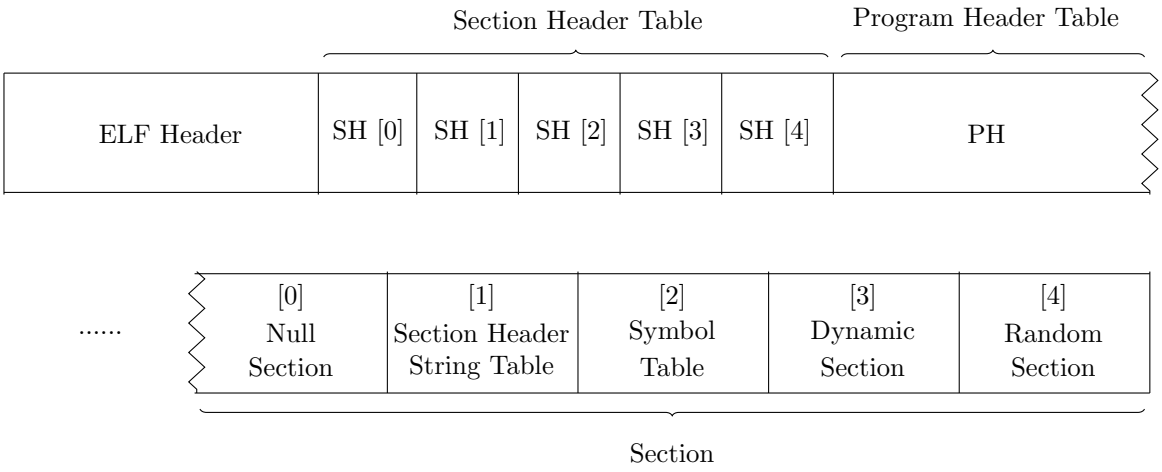


Figure 3.4: DASE's ELF layout. SH is Section Header, and PH is Program Header. Numbers in brackets are array indices.

> After locating the possession modifier, DASE applies GR1 to resolve the properties of the head noun. In this example, GR1 will return the field name "sh_type" as the property of "field".

If a comment only specifies a partial field name, DASE will resolve the name into a fully qualified name. For example, the comment "Legal values for e_type" specifies a field name "e_type" without the struct name. DASE maps this field name to structs that contain this field name and generates the fully qualified names, "Elf32_Ehdr→e_type" and "Elf64_Ehdr→e_type".

In addition, DASE extracts constraints from manual pages using regular expressions. Manual pages often show a `struct` declaration, followed by the constraints (if available) for each field in the `struct`. The valid values for each struct field can be identified based on the indentation of the manual page. Based on this layout, DASE first locates the name of the `struct`, and maps it to each of the constraints that are listed below it. The output of this analysis is also a list of constraints that can be directly used by the symbolic execution part of DASE.

3.4.2 Adding File Layout Constraints

ELF files follow a certain layout, which also defines valid ELF files. Therefore, in addition to extracting the file format constraints as described in Section 3.4.1, we add file layout

constraints for ELF by reading the ELF specification [153]. Our results show that both the file format and file layout constraints contribute to the improvement of DASE.

An ELF file always starts with an *ELF header* followed by the two header tables, *section header table* (SHT) and the *program header table* (PHT). SHT contains an array of *section header*s, PHT contains an array of *program header*s, and object files' real data are in the *section*s. In order to reduce the workload of the constraint solver and focus on important parts of ELF, we adopt a rigid layout as shown in Figure 3.4. SHT is set to have five section headers. The first section (at index 0) is a null section, followed by a string table, symbol table, dynamic section, and random section. The second and fifth section are set with a size of 8 bytes, and the third and fourth section are set with a size that is enough to hold two symbols. PHT is set to contain one random program header.

Note that our ELF layout is incomplete. We retain this incompleteness to give DASE the ability to explore close-to-valid inputs to explore boundary cases. In addition, input constraints can be slightly relaxed to include more close-to-valid inputs, which remains as our future work.

3.4.3 Extracting Valid Options

We automatically extract valid options only from manual pages because we find that code comments do not describe valid options. Since manual pages list the valid options in a standardized format, our parsers perform simple regular expression matching, which is effective and accurate. DASE takes a manual page as input and outputs a list of valid command line options. It uses two regular expressions, one for *short* options (a single dash followed by a single letter), and one for *long* options (two dashes followed by multiple letters). If a short option has a long option equivalent, DASE keeps only the short option.

We also explored the extraction of valid options directly from the source code using Clang tools [154]. However, since Clang often undergoes frequent version changes that break the API compatibility of the Clang tools and applications may contain different parser code, we decide to utilize manual pages to extract the constraints. Although manual pages might not always contain the most update-to-date information, the extraction code can extract valid options from a wider range of programs with less effort which makes it a better choice.

3.4.4 Using Options to Flatten Symbolic Execution

DASE takes the options extracted in Section 3.4.3 to trim and reorganize the dynamic symbolic execution tree as shown in Figure 3.3. Specifically, instead of having s symbolic arguments, DASE runs the program with $s-1$ symbolic arguments and a concrete valid option, which forms one execution branch. In this way, DASE creates n execution branches for a program with n valid options (one for each valid option). The aim is to balance the testing effort on each option (and the corresponding functionality), which should be of the similar semantic importance (at least not as skewed as $\frac{1}{2}$ versus $\frac{1}{2^n}$ as shown in Section 3.2). The generated branches are then prioritized by search strategies. We can consider this technique as a "partition" of the execution tree. *The $s-1$ arguments remain symbolic, which can expand to any concrete options. Therefore, it is possible to cover combinations of command-line options such as "`-r -f`" (of `rm`) in our approach.* To ensure the completeness of this "partition", we add a branch for an invalid option and a branch for a null option.

3.5 Evaluation Method

We use three coverage criteria reported by `gcov`, i.e., line, branch, and call coverage (% of executed function calls), as our main coverage metrics. The coverage criteria and `gcov` are widely used in literature [40, 53, 41, 54].

3.5.1 Evaluated Programs

We evaluate DASE on the following 88 programs from 5 popular and mature fundamental software suites for Unix-like systems. The sizes of these programs are at the *same scale* as the ones evaluated by previous work [53, 60, 41, 54].

Coreutils 6.10. Coreutils, also evaluated by KLEE, is a package of GNU programs that consists of basic file, shell, and text manipulation utilities.

`diff` 3.3. `diff` compares files line by line and outputs the differences.

`grep` 2.18. `grep` searches files for given patterns.

`objdump` & `readelf`(b) 2.24. `objdump` and `readelf` are used for displaying the contents of ELF files. Since both Binutils and elftoolchain contain a `readelf` program, we use `readelf`(b) to denote the `readelf` program in Binutils and `readelf`(e) to denote the one in elftoolchain.

`elfdump` & `readelf(e)` **r2983**. In order to test our ELF model more thoroughly, we select ELFTOOLCHAIN's counterparts for the above two programs. ELFTOOLCHAIN provides similar tools as BINUTILS, but favors well-separated and well-documented libraries.

3.5.2 Experimental Setup

All automatically extracted file format constraints and valid options (without manual examination for zero manual effort) are used as input constraints for all programs when applicable. DASE extracts file format constraints for ELF and uses them for the 4 ELF processing programs (`objdump`, `readelf`(b), `elfdump`, and `readelf`(e)) for path pruning. ELF is a boardly used main standard for binaries in Unix-like systems. *One can use the ELF model that we build to potentially improve test generation for all programs that read or write ELF binaries on a Unix-like platform.* In addition, DASE extracts valid options for the rest of the programs automatically and uses them to guide the symbolic execution on them.

To show the generality of our techniques of automatically extracting file format constraints, DASE extracts file format constraints for two additional standard file formats—Tar from `tar.h`, and the Common Object File Format (COFF) from `coff/internal.h`.

We run KLEE and DASE on each program until *no new instructions are covered in a certain amount of time*: 15 minutes for COREUTILS programs and 30 minutes for the rest due to their larger sizes. *This stop criterion allows both DASE and KLEE to run until they cannot make progress in coverage in a fixed time period, which is similar to that of the previous paper [54], but different from that of KLEE [40], in which each program is only allowed to run for one hour. In our experiments, the actual run time of each program varies from 6 seconds to 11.5 hours. We have also conducted experiments using the stop criterion from the KLEE paper, and DASE still achieves a similar amount of improvement over KLEE.*

The other parameters are set by following the instructions from KLEE's authors [155]. The key parameters are:

```
klee PROG -sym-args 0 1 10 -sym-args 0 2 2
          -sym-files 1 8 -sym-stdout
```

where `PROG` is a program in COREUTILS. While for DASE, we keep all the parameters the same as for KLEE, except for replacing a symbolic argument with a list of valid options.

For `diff` and `grep`, we set the symbolic file size to 100 bytes because they are meant to process textual files.

For the ELF processing programs, we use the following parameters respectively for KLEE and DASE:

```
klee -sym-args 0 2 2 -sym-files 1 640
klee -sym-args 0 2 2 -sym-elfs 1 640
```

where `-sym-elfs` holds our ELF model described in Section 3.4.2.

We conduct our experiments on an Intel Core i5-2400 3.10GHz CPU machine running Ubuntu 13.10. KLEE is built from git revision `a45df61` with LLVM 2.9.

3.6 Evaluation Results

This section shows that DASE finds more previously unknown bugs, improves code coverage on top of different search strategies, complements developer tests, and extracts input constraints automatically. We also show that our results are statistically significant.

3.6.1 Detected Bugs

Using the constraints automatically inferred from documents (without any manual verification), DASE finds more bugs than KLEE. KLEE detects 3 previously unknown bugs from the 88 programs while DASE can uncover 13 previously unknown bugs (KLEE failed to detect 12 of them). Table 3.1 lists all of the detected previously unknown bugs.

DASE found 2 previously unknown bugs in COREUTILS and 3 in BINUTILS (`objdump` & `readelf`(b)), both of which have already been thoroughly tested by many symbolic execution tools. For example, COREUTILS has been tested by Veritesting [53], ZESTI [60] and KLEE [40], and BINUTILS has been tested by Veritesting [53], ZESTI [60], and KATCH [41]. Finding 5 new bugs in those extensively-tested suites demonstrates DASE's ability in finding new bugs and improving symbolic execution.

`readelf`(b) fails with segmentation fault when the input file contains malformed attribute sections (of type `SHT_ARM_ATTRIBUTES`) [156]. The bug exists in the function `process_attributes()`, which is shown in Figure 3.5. Pointer `p` walks through the whole section. At line 19, 4

Table 3.1: New bugs detected by KLEE and DASE. "✓" denotes a bug is found by a tool. "IU" means "Integer Underflow." "DBZ" is "Divide By Zero." "IL" is "Infinite Loop." "NPD" means "NULL Pointer Dereference." "POB" stands for "Pointer Out of Bounds." "ME" is "Memory Exhausted."

No	Program	Location	Problem	KLEE	DASE
1	`readelf`(b)	`readelf.c`:12202	IU		✓
2	`objdump`	`elf-attrs.c`:463	IU		✓
3	`objdump`	`elf.c`:1351	POB		✓
4	`readelf`(e)	`readelf.c`:4015	DBZ		✓
5	`readelf`(e)	`readlef.c`:2862	DBZ		✓
6	`readelf`(e)	`readelf.c`:3680	DBZ		✓
7	`readelf`(e)	`readelf.c`:3930	IU		✓
8	`readelf`(e)	`readelf.c`:3961	IL		✓
9	`readelf`(e)	`readelf.c`:4102	IL		✓
10	`readelf`(e)	`readelf.c`:2662	NPD		✓
11	`readelf`(e)	`readelf.c`:2426	POB	✓	
12	`elfdump`	`elfdump.c`:1509	POB	✓	✓
13	`elfdump`	`elf_scn.c`:87	POB	✓	
14	`head`	`head.c`:207	ME		✓
15	`split`	`split.c`:333	ME		✓

bytes are read and interpreted as the length (`section_len`) of the subsequent data structure. Directly after that, the program expects to read a string and assign its length to `namelen`. However, `section_len` can be a number smaller than `namelen + 4`, which causes an integer underflow at line 24. The variable `section_len`, which becomes an extremely big number after underflow, is later used as the stop condition of a continuing reading of the following memory, which eventually causes a segmentation fault.

Five other functions are ahead of `process_attributes()` in the call stack, namely, `main()`, `process_file()`, `process_object()`, `process_arch_specific()`, and `process_arm_specific()`. Each function reads and processes specific parts of the input ELF file. For example, to correctly invoke `process_attributes()`, the condition for the `if` statement at lines 2–5 must evaluate to false. The automatically extracted ELF constraints guide DASE to generate an ELF file that satisfies all these constraints to reach `process_attributes()` and expose the bug. This close-to-valid ELF file helps DASE detect this bug. `readelf`(e) contains a similar bug.

```
1  static int process_file_header(void) {
2    if (elf_header.e_ident[EI_MAG0] != ELFMAG0
3     || elf_header.e_ident[EI_MAG1] != ELFMAG1
4     || elf_header.e_ident[EI_MAG2] != ELFMAG2
5     || elf_header.e_ident[EI_MAG3] != ELFMAG3) {
6        error (_("Not an ELF file - ..."));
7        return 0;
8    } ...
9  }
10 ...
11 static int process_object(...) { ...
12   if (!process_file_header())
13      return 1; ...
14   process_arch_specific(file); /* calls
15      process_attributes() indirectly */ ...
16 }
17 ...
18 static int process_attributes(...) { ...
19   section_len = byte_get(p, 4);
20   p += 4;
21   ...
22   namelen = strlen((char *)p) + 1;
23   p += namelen;
24   section_len -= namelen + 4;
25
26   while (section_len > 0)
27   ...
28 }
```

Figure 3.5: Buggy code in `readelf.c` from BINUTILS.

The `head` program fails with memory exhaustion when invoked with options `-c -1P`, which tells `head` to print all but the last `1P` bytes of the input file. Since `P` is a large unit of 1024^5, `head` tries to allocate a large amount of memory, which exceeds the total amount of available memory. According to the comment, `head` is not expected to "fail (out of memory) when asked to elide a ridiculous amount". For bigger units (e.g., `Z` and `Y`), `head` exits with the correct error message—"number of bytes is so large that it is not representable". Neither developers' hand-written tests nor KLEE generated tests detect this bug.

Two bugs can be found by KLEE but not by DASE due to the following reason. The ELF file to trigger the bugs has a very large `e_shoff` value (SHT's offset from the beginning of the EFL file), which is incompatible with our ELF model. As shown in Section 3.4.2, we manually fixed the offset to layout the SHT. Missing these two bugs shows the tradeoff involved in designing the ELF model. DASE focused on those more valid inputs to test the core logic.

Table 3.2: Coverage results with KLEE's default search strategy. "Line", "BR", and "Call" show the total number of executable lines of code (ELOC), branches, and calls for each program, reported by `gcov`. "K" stands for KLEE and "D" is DASE. "Δ" is the improvement in percentage points of DASE over KLEE.

Program	**Line**	**K**	**D**	**Δ**	**BR**	**K**	**D**	**Δ**	**Call**	**K**	**D**	**Δ**
		%	%	**pp**		%	%	**pp**		%	%	**pp**
COREUTILS	18329	66.2	75.6	**+9.4**	12674	69.9	77.3	**+7.4**	7008	56.6	67.5	**+10.9**
`diff`	526	59.1	67.9	**+8.8**	489	68.1	69.7	**+1.6**	150	46.6	59.3	**+12.7**
`grep`	932	37.3	58.4	**+21.1**	786	40.3	59.2	**+18.9**	266	33.5	53.6	**+20.1**
`objdump`	1687	19.4	25.6	**+6.2**	1270	16.9	22.8	**+5.9**	463	16.6	19.4	**+2.8**
`readelf`(b)	6998	6.9	15.2	**+8.3**	5410	6.2	16.6	**+10.4**	1959	6.9	13.5	**+6.6**
`elfdump`	1539	16.1	22.1	**+6.0**	1157	20.4	30.7	**+10.3**	533	16.5	23.6	**+7.1**
`readelf`(e)	3571	13.0	28.0	**+15.0**	2550	18.5	34.5	**+16.0**	1126	10.8	25.4	**+14.6**

Our results clearly demonstrate the benefits of our design choice: DASE finds 10 more bugs than KLEE. One can relax the constraints to explore fewer valid inputs and potentially cover these two bugs. Running KLEE and DASE together to gain benefits from both is also a good solution.

3.6.2 Code Coverage

Table 3.2 shows the overall code coverage achieved by KLEE and DASE. DASE outperforms KLEE on the 88 programs: it increases the line coverage, branch coverage, and call coverage by 14.2–120.3%, 2.3–167.7%, and 16.9–135.2% respectively, which are 6.0–21.1 pp, 1.6–18.9 pp, and 2.8–20.1 pp increases. For example, the line coverage boost on `grep` is 21.1 pp. Programs `readelf`(b), `objdump`, `readelf`(e), and `elfdump` are difficult to test because their inputs involve the complex ELF format. Despite the lower coverage, DASE detected new bugs in them that existing techniques did not detect as shown earlier. Figure 3.6 shows the coverage improvement of DASE over KLEE on `readelf(b)` over time. It shows that the improvement increases as time proceeds.

The coverage percentages for COREUTILS are different from those of the KLEE paper [40]. The difference is inevitable because the KLEE tool has evolved significantly since then, including major code changes of KLEE (e.g., removals of special tweaks), and an architecture change from 32-bit to 64-bit. We choose the latest version of KLEE at the time of the experiment because the original version used in the KLEE paper is not publicly available. For a fair comparison, the configurations for KLEE and DASE are identical.

DASE explored deeper than KLEE. Since DASE filters out "uninteresting" paths, we

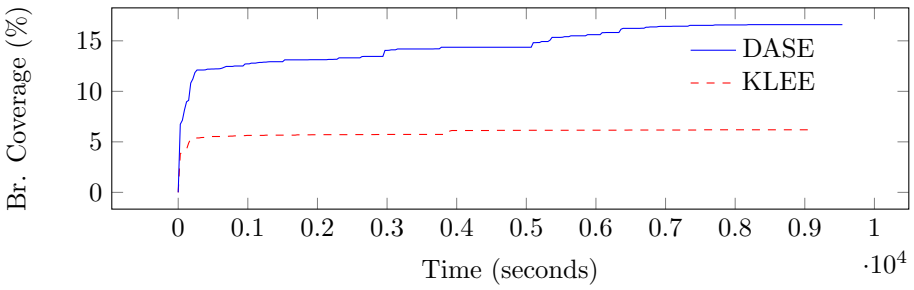


Figure 3.6: Branch coverage on `readelf(b)` over time.

Table 3.3: Number of instructions of generated test cases, showing that DASE explored deeper than KLEE. "K-" stands for KLEE and "D-" stands for DASE. "AVG-I" and "MAX-I" is the average and maximum number of instructions for the generated test cases respectively. Since COREUTILS includes multiple programs, a range (the minimum and the maximum) is shown.

Program	**K-AVG-I**	**D-AVG-I**	**K-MAX-I**	**D-MAX-I**
COREUTILS	7132	8170	11682	13200
(82 programs)	49688	55672	320138	1189470
`diff`	18483	22427	35432	35976
`grep`	25942	26231	43424	53081
`objdump`	45915	62236	104479	206874
`readelf`(b)	11570	17800	24884	36196
`elfdump`	13827	28560	24433	319744
`readelf`(e)	18009	30069	29140	61233

expect it to explore deeper. We count the number of executed instructions for each test case generated by KLEE and DASE to approximate the depth of the corresponding paths. The average and maximum numbers are shown in Table 3.3, which shows that DASE generates test cases with much more instructions executed, indicating that DASE goes much deeper into the execution tree than KLEE. For the ELF processing programs, both the averages and maximums almost double their counterparts of KLEE. The difference is expected because while KLEE is still exploring at the early stage of the ELF sanity check, DASE has already penetrated through that part with the help of our ELF model.

DASE covered more functionalities and options than KLEE. To investigate DASE's coverage gain in detail, we manually check the coverage difference of KLEE and DASE on

Table 3.4: Coverage of combining DASE with developer test cases, showing that DASE complements developer tests. "V" is developer tests. "D" is DASE combined with developer tests. Do note that `readelf(e)` has no developer test cases hence is missing.

Program	Line	V %	D %	Δ pp	BR	V %	D %	Δ pp	Call	V %	D %	Δ pp
COREUTILS	18332	66.1	84.4	+**18.3**	12670	73.2	87.2	+**14.0**	7008	53.2	73.5	+**20.3**
`diff`	526	57.0	81.0	+**24.0**	489	72.2	87.3	+**15.1**	150	50.7	71.3	+**20.6**
`grep`	932	82.0	86.5	+**4.5**	786	87.0	89.8	+**2.8**	266	73.7	81.2	+**7.5**
`objdump`	1687	58.6	64.3	+**5.7**	1270	66.9	68.9	+**2.0**	463	51.2	57.2	+**6.0**
`readelf`(b)	7038	28.8	33.5	+**4.7**	5424	43.5	46.5	+**3.0**	1964	28.6	33.2	+**4.6**
`elfdump`	1084	71.6	75.0	+**3.4**	813	86.5	86.5	+**0.0**	264	46.2	52.9	+**6.7**

`diff`. KLEE explores only 27 out of the 55 distinct options[5], which are the shallower options, while DASE covers 46 options. The result agrees with our analysis in Figure 3.3. We have similar observations for the ELF processing programs. We manually examine the coverage difference on `readelf.c` (BINUTILS). For the three functions related to dynamic section, `*_dynamic_section()`, in which `*` means `get_32bit`, `get_64bit`, or `process`, KLEE fails to cover any of them, while DASE naturally tests them all because our ELF model has a dynamic section. In addition, many other functions, such as `print_symbol()`, are missed by KLEE but covered by DASE.

3.6.3 DASE Complements Developer Tests

Since automated test generation aims to complement developer generated tests, we evaluate whether DASE finds bugs that developer tests cannot detect, and improves code coverage on top of developer tests. DASE *detected a total of 13 bugs on the evaluated programs that developer generated tests fail to detect* (Section 3.6.1). Table 3.4 shows the coverage comparison. We can see that by adding DASE generated tests, the line coverage is improved by 3.4–24.0 pp. Together with Table 3.2, we can see that for COREUTILS and `diff`, DASE alone can generate tests to achieve comparable code coverage as developer generated ones. Although the coverage improvement on `objdump`, `readelf`(b), and `elfdump` is relatively small, the DASE generated tests detected previously unknown bugs for all of them. The results demonstrate that DASE can be used by developers to find more bugs and further improve testing coverage even if manual tests exist.

[5]We count options that invoke the same code segment as one option.

3.6.4 Constraint Extraction Results

DASE automatically extracted input constraints from manual pages and comments for command line options and three file formats with accuracies of 97.8–100%.

Specifically, for command-line options, DASE automatically extracted 776 valid options from manual pages: 683 from the 82 COREUTILS programs, 46 from `grep`, and 47 from `diff`. The accuracy is 100%.

For ELF processing programs, we manually enforced 63 constraints to form the layout of our ELF model shown in Figure 3.4. By analyzing the ELF header file and ELF manual page, DASE automatically extracts 60 values for 16 constraints regarding array index-value pairs, and 312 values for 20 constraints regarding valid field values. For example, the constraint "`assume( Elf32_Shdr→e_type == 0 | Elf32_Shdr→e_type == 1);`" is one constraint with two values (0 and 1). In the case where a constraint exists from both the header file and manual page, DASE combines all the values within both documents and creates a single new constraint with all the merged values. The ELF header file constraints is a superset of the manual page constraints except for two constraints, which specify valid values for the `EI_VERSION` indices of the `e_ident` arrays. The accuracy of the extracted constraints is 97.8%.

The breakdown of the constraints extracted from the header file and the manual page is as follows. From the manual page, DASE automatically extracts 46 values for 16 constraints regarding array index-value pairs, and 72 values for 8 constraints regarding valid field values. The accuracy is 100%.

From the header file, DASE extracted 56 values for 14 constraints regarding array index-value pairs, and 312 values for 20 constraints regarding valid field values. Among the 312 values for 20 constraints regarding valid field values, 8 values are invalid, which affect 8 constraints. The accuracy is 97.8%. The inaccuracy results from a special kind of macro, `*_NUM`, in `elf.h`. This macro represents the total number of valid values, which is not a valid value. Among all the constraints, 10 constraints (consisting of 56 values) on special section types are not used because they are not applicable to our model. We can incorporate them when we improve our ELF model in the future.

DASE also extracted constraints from Tar and COFF's header files. It extracted 23 values for 2 constraints for the Tar file format, and 18 values for 2 constraints for the COFF file format. All of the extracted constraints are correct.

Impact of Incorrect Constraints. To understand the impact of incorrect constraints, we ran DASE with only the correct constraints (our main evaluation applies all constraints

to minimize manual effort). The coverage and bug finding results are almost identical, suggesting that DASE is robust when a few incorrect constraints are provided.

Potential Effort Savings. Automated constraint extraction is important yet challenging [146, 147], and much work has been proposed to infer constraints from source code and execution traces automatically [157, 158, 159]. The proposed automated constraint extraction technique (takes 10–60 seconds to run) can save the effort of manually writing constraints. It is beneficial to automate the constraint extraction process to keep the constraints up to date since ELF, Tar and COFF file formats all have many revisions since their standardization.

DASE extracted almost all constraints in the header files and manual pages. This can be expanded by analyzing more comprehensive file specifications such as the ELF specification [153]. In the future, we would like to extend the proposed NLP techniques to analyze other formats, e.g., TCP/IP packets and XML format.

3.7 Threats to Validity

We discuss the different threats to validity to our experiments.

3.7.1 Internal Validity

Internal validity concerns the extent where one can claim the cause and effect between variables in the experimental design.

Our evaluation currently does not distinguish between the contribution of the two categories of input constraints. Therefore, it is not possible to have a deeper understanding of the impact of these two variables on the results.

3.7.2 External Validity

External validity concerns if the conclusions can be generalized outside of the study.

The evaluation currently only includes five software suites. Although Binutils and ELF toolchain are included to test the ELF file format, additional software suites should be included to ensure the results are generalizable to other file formats.

While the natural language processing techniques are effective on the three evaluated file formats, the techniques may not generalize to other types of documentation. New grammar rules may be needed to support new types of sentence structures. In addition, the accuracy and quality of the extracted constraints depend on the quality of the documentation.

3.7.3 Construct Validity

Construct validity concerns if a test measures the intended construct of the study.

The evaluation measures how deep the symbolic execution tool had explored based on the number of instructions that are covered by the generated test cases. There may be other more accurate ways to measure the value.

3.7.4 Conclusion Validity

Conclusion validity concerns the if the reached conclusions about the relationships in the data are reasonable.

The evaluation concluded that there are potential effort savings from the automated constraint extraction tool. Although the constraint extraction process is automated, we still have to define the file layout constraints which requires manual effort from reading and understanding the ELF specifications. A more controlled study may be needed to factor in the amount of time that it takes to learn a file format.

3.8 Summary

This work presents Document-Assisted Symbolic Execution (*DASE*)—a novel and general approach to extract input constraints from documents automatically to improve symbolic execution for automated bug detection and test generation. DASE prunes and flattens paths based on their *semantic* importance to help search strategies prioritize execution paths more effectively. DASE detected 12 previously unknown bugs that KLEE fails to detect, 6 of which have been confirmed by the developers on 88 mature programs. Compared to KLEE, DASE increases line coverage, branch coverage, call coverage by 6.0–21.1 pp, 1.6–18.9 pp, 2.8–20.1 pp, respectively. In the future, it would be promising to negate the input constraints to focus on testing error handling code.

Chapter 4

Documentation Analysis: Automatic File Repair

4.1 Motivation

Corrupted files cause many software failures which impacts software reliability. Specifically, file corruption causes software crashes [15], security issues [16, 17, 18, 19, 20], users unable to access valuable data, etc. For example, a corrupted PDF file triggered a null pointer bug in the Google Chromium browser, which caused the browser to crash [15]. Files often become corrupted due to problems such as file system bugs [21, 22], network transfer errors [23], malicious modifications of a file [24], and buggy software [25].

It is important to repair corrupted files to mitigate the resulting software failures. We propose to utilize documentation constraints that are extracted from the user manual for the repair of corrupted PDF files. The repair of corrupted files is challenging [26, 27]. File viewers often fail to display corrupted files because corrupted files do not follow PDF specifications and cannot always be parsed by the viewer. Thus, practical techniques to repair corrupted files are in high demand.

Designing a repair tool requires a good understanding of the corrupted file problem. It is important to understand the number of real-world corrupted files in the wild; the causes of file corruption (e.g., corrupted base structure and corrupted font); the impact of file corruption (e.g., incorrect functionality, crash, and performance degradation); and whether existing techniques can repair them. All these questions need to be investigated to design a solution for repairing corrupted PDF files.

There is neither a comprehensive study to quantify the existence of corrupted files in the real world, nor a study to understand their impact. There are some anecdotal evidence on the existance of corrupted files in the real world [26], and previous work analyzed the impact of data corruption in storage systems [28, 29, 30]. Force Open [27] generated a large corpus of corrupted PDF files using a testing technique called fuzzing [160, 145, 161, 162] to test their PDF file repair technique. However, the fuzzing technique generates corrupted files by overwriting a small number of bytes with random data until the file cannot be opened by a PDF reader, `pdftotext` [163], which may not be representative of real-world corrupted files.

In general, there are two types of corrupted files:

(1) Real-world corrupted files: these documents became corrupted unintentionally, e.g., due to network transfer errors, bugs in file creators or file processing software. These documents contain valuable content.

(2) Purposely corrupted files: these documents were corrupted on purpose, e.g., malicious modification to expose bugs and security vulnerabilities in a file viewer. Typically, users do not care about the content of these documents.

It is beneficial to repair both types of documents to avoid failures of file viewer and file creator software including crashes and security vulnerabilities. For real-world corrupted files, file repair has an additional benefit of allowing users to view valuable content in the file such as medical prescriptions, tax documents, etc.

While we study and repair both types of files in this work, our focus is on real-world corrupted files as they have not been studied before and contain important content to recover. Specifically, we conduct a case study to understand and repair corrupted PDF files, since the PDF file format is one of the most popular electronic file formats [164, 165]. First, we collect the first dataset of real-world corrupted PDF files, understand the causes and impact of corrupted PDF files, and study the repair capabilities of existing PDF repair tools. Second, based on the findings from the empirical study, we present a white-box approach, *DocRepair*, for repairing corrupted PDF files.

A fundamental problem of existing file repair tools is that they only use hardcoded heuristics to validate the correctness of a corrupted file. For example, Listing 4.1 shows the corrupted part of a PDF file that causes Google Chrome browser to crash [15], where the file cannot be repaired by existing tools. The corrupted value, -406081386, on line 4 refers to an object number in the PDF file format. The PDF file format specification [166] specifies that the value is an indirect reference and it must be a positive integer object number. The corrupted value, -406081386, is not a valid object number that exists inside

```
1 trailer
2 <<
3 /Size 38
4 /Root -406081386 1631658500 R <- invalid
5 /Prev 33749
6 >>
```

Listing 4.1: Corrupted part of a PDF file causing a crash in Chromium bug #134551 [15].

the PDF file, which eventually leads to a null pointer crash in the PDF viewer because the Chrome browser parses the PDF file without validating the data field's correctness.

There are several common PDF repair tools for repairing corrupted PDF files such as Mutool [73], PDFtk [74], and GhostScript [78]. A repair tool accepts a corrupted PDF file as the input and produces a repaired file as the output. A corrupted file is defined to be successfully repaired by the repair tool if a PDF viewer can open and display the content of the repaired file. Existing PDF repair tools [73, 74, 78] failed to recover the content of the file because none of the existing repair tools check if the corrupted data field contains an object number that points to a valid object in the PDF file.

Learning from our study of real-world corrupted PDF files, we designed *DocRepair* to mitigate the crash and recover the content of the file with a set of new repair operators. The repair operators detect missing and invalid objects for specific components within the PDF file, which is done by analyzing the references between objects (i.e., object numbers) and the validity of the data structure (i.e., values). If an issue is detected on a specific component of the file, the repair operator attempts to recover the object and data structure, which enables PDF viewers to open the repaired file and display the file's content successfully after the repair.

Given the corrupted file described in Listing 4.1, which triggers a bug in Chrome, we applied three existing repair tools (Mutool [73], PDFtk [74], and GhostScript [78]) and DocRepair to attempt to repair it. Figure 4.1 shows the screenshots of PDF files when opened with the buggy version of Chrome. Figure 4.1a shows the original corrupted file from the bug report, and Figure 4.1b, Figure 4.1c, and Figure 4.1d show the repaired version of the corrupted file by Mutool, GhostScript and DocRepair. Mutool generated a repaired file that triggers an error message, "Failed to load PDF document," in Chrome (Figure 4.1b). PDFtk failed to generate a repaired file. GhostScript generated an empty PDF file that contains no content (Figure 4.1c). Although existing tools can prevent the crash of the Chrome tab, DocRepair generated a better repair by recovering the valuable

content (Figure 4.1d). After the developers fixed the bug in Chrome, while, the corrupted file does not crash Chrome anymore, DocRepair is still the only tool that produces a repaired file that can be correctly displayed.

This work answers four research questions (RQ), where RQ1–RQ3 are part of our empirical study of real-world corrupted PDF files, and RQ4 evaluates the effectiveness of DocRepair against existing work.

RQ1: *How many real-world corrupted PDF files exist and what is their impact?*

Corrupted PDF files can have a negative impact on end users such as program crashes and security vulnerabilities. Therefore, we want to understand (1) how many real-world corrupted files exist in the wild, (2) if these files contain any security threats to the end user, and (3) if these files have an impact to the end user and target software.

RQ2: *Can existing repair tools repair corrupted PDF files?*

The question attempts to understand the repair capability of existing file recovery tools. The demand for a new repair tool exists if many corrupted PDF files cannot be repaired by any existing tools.

RQ3: *What types of real-world file corruption exist?*

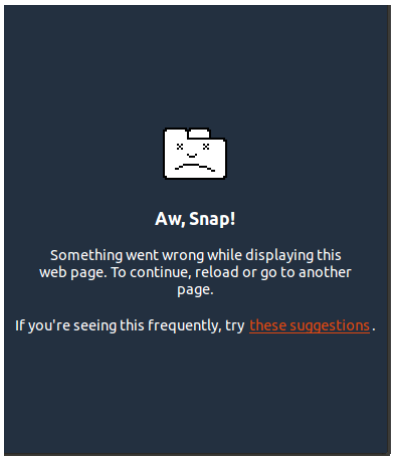
The question identifies the causes of corrupted files such that we can incorporate the common scenarios into the proposed repair algorithm.

RQ4: *How does DocRepair compare to existing file recovery tools?*

We evaluate the ability of our new repair technique, DocRepair, at repairing both real-world and purposely corrupted files that existing tools cannot repair.

This work makes the following contributions:

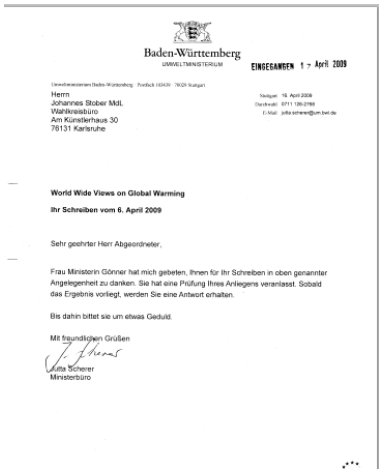
- We created the first dataset of 119 real-world corrupted PDF files. We complement the dataset with 198 purposely corrupted files that trigger bugs and vulnerabilities in PDF readers. This dataset will help researchers to better understand real-world file corruption and practitioners to further develop PDF repair tools.



(a) Corrupted file that crashes Chrome tab.

(b) File repaired by Mutool. No content is recovered.

(c) GhostScript output an empty page.



(d) File successfully repaired by DocRepair.

Figure 4.1: Corrupted and repaired versions of the corrupted file opened on Chrome 21.0.1180.89 (Chromium bug #134551). The corrupted version of the file crashes the browser tab.

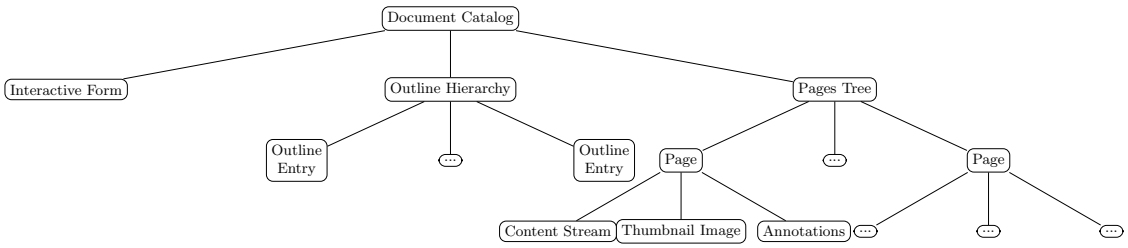


Figure 4.2: PDF body's document tree structure where each node represents an object.

- We perform an empirical study on the repair capability of existing repair tools and the common causes of PDF file corruption. Our results show that 63 of the real-world corrupted files and 105 of the purposely corrupted files cannot be repaired by existing tools. A manual inspection of real-world corrupted files revealed that the most common causes of PDF file corruption include corruption in base structure, data stream, font resource, page tree structure, and data field value.
- We propose DocRepair, a new automatic file repair technique which uses a two-stage algorithm that contains seven repair operators. DocRepair repaired a total of 30 of the 119 real-world corrupted PDF files, 7 of which cannot be repaired by existing tools. We also propose to combine multiple repair tools as DocRepair+ and show that the combined tool can repair 63 of the 119 real-world corrupted files. Existing repair tools including Mutool, PDFtk, and GhostScript repaired a total of 38, 16 and 45 of the 119 real-world corrupted files respectively.

4.2 Background

4.2.1 PDF File Format

PDF is a file format that was created to share documents across platforms. It was opened up as ISO standard 32000-1 in 2008 [167]. The PDF file format contains four major components: *header*, *body*, *cross-reference table*, and a *file trailer*.

- The *header* consists of the keyword, `%PDF-` followed by a version number between 1.0 and 1.7, and at least four high bit ASCII characters.

```
1 trailer
2 <</Size 38/Root -406081386 1631658500 R>>
3 ====
4 trailer
5 <</Root 36 0 R /Size 38>>
```

Listing 4.2: An incorrect repair generated by Mutool (top) and correct repair generated by DocRepair (bottom).

- The *body* contains a list of objects that represent the contents of the file. Figure 4.2 shows the body as a document tree structure, where each node represents an object. For example, the `pages tree` object references a list of `page` objects, and each `page` object has a reference to a `content stream` object that stores the text and layout of the page.
- The *cross-reference table* contains a list of pointers (positional offsets) that point to the location of each object in the file.
- The *file trailer* is the entry point for parsing a PDF document, which allows a reader to locate the *cross-reference table* and the key objects to parse the PDF file.

4.2.2 Existing Repair Approaches

Existing tools including Mutool [73], PDFtk [74], and GhostScript [78] repair PDF files through a rewriting process. They would parse the corrupted file into memory (if the corrupted bytes do not cause cascading parsing failure) and write the parsed content back out with a new syntax as the repaired file. The approach allows existing repair tool to detect corruption that impacts the syntactical of the file and discard them, and detect missing components within the PDF file structure (Section 4.2.1) for reconstruction. However, existing tools do not perform deep validation on the parsed data values. For example, in the motivating example (Listing 4.1), Mutool failed to detect file corruption because it only checks if the file trailer is syntactically correct. Since the file trailer is syntactically correct, Mutool copied the semantically incorrect values to the output PDF file that is supposed to be repaired. DocRepair is the only repair tool that is able to detect the incorrect value and infer the correct value to repair the file (Listing 4.2).

4.3 Definitions

In this work, a file is *corrupted* if it does not conform to the standard file format specification. Existing repair tools attempt to detect violations of the file specification and repair them using repair operators (RO). A repair operator's purpose is to address specific issues of the PDF file. For example, most repair tools deploy repair operators for the malformed file header, file trailer and cross-reference table (Section 4.2).

For any corrupted file where the ground truth (correct file) is available, we consider a file *fully repaired* if the generated file from a repair tool is identical to the correct file. Since the corresponding correct files for most real-world corrupted PDF files are unavailable, following prior work on PDF file inconsistency detection [168] and black box file repair [27], we define a file as *repaired* if no more than one PDF viewer fails to display some content from the PDF file. For example, if only one viewer fails to display the contents of a PDF file, but all other PDF viewers (six for this study) displays it successfully, we consider the PDF not corrupted to accommodate cases where the one viewer contains a bug which is common as shown in prior work [168].

4.4 A Study of Corrupted PDF Files

While the problem of real-world corrupted PDF files is known, there has only been anecdotal evidence of its existence [26]. Therefore, a study to better understand the extent of the problem and the most common causes and impact of file corruption is needed. The purpose of the study is threefold: (1) obtaining empirical evidence of how many documents in the wild are corrupted and their impact (RQ1), (2) checking if existing repair tools can repair real-world corrupted PDF files (RQ2), and (3) understanding the causes of file corruption (RQ3).

We start the empirical study by collecting potentially corrupted PDF files from multiple bug tracking systems (Section 4.4.1). Since bug tracking systems contain a large number of files, many of which are not corrupted, we develop an automated tool—`CorruptCheck`—to identify corrupted PDF files automatically (Section 4.4.2), which are then manually inspected. We present the PDF repair tools and viewers used by CorruptCheck (Section 4.4.3) and show the findings from our empirical study (Section 4.4.4).

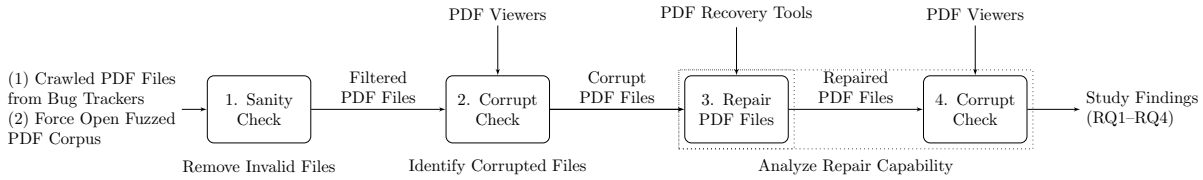


Figure 4.3: Overview of the empirical study and evaluation.

4.4.1 Collecting Corrupted PDF Files

Real-World Corrupted PDF Files To find real-world corrupted PDF files in the real world, we crawl files from existing software bug tracking systems. We focus on bug trackers because they often have corrupted files attached to bug reports to help reproduce bugs. Although we only collect PDF files from eight bug trackers, the bug trackers cover a large number of software applications that processes PDF files. Therefore, we developed a web scraper using a web-crawling framework called Scrapy [169] to extract PDF files from bug report attachments. We also attempted to use an existing government document dataset crawled by previous work [170], but the previous investigation seems to indicate that it does not contain corrupted PDF files [168]. Therefore, we do not use the government document dataset in the study. The web scraper returns a list of PDF files, which we later manually inspect in the empirical study to identify the real-world corrupted PDF files.

The study collected PDF files from eight bug trackers including GNOME, Mozilla, Chromium, LaunchPad, GhostScript, KDE, Apache and Freedesktop (Table 4.2). Although we only collect PDF files from eight bug trackers, the bug trackers cover a large number of software applications that process PDF files. We selected these bug tracking system based on two criteria. (1) We include all the evaluated PDF viewers' bug tracker whenever available (Launchpad tracker for Evince, Qpdfview and Xpdf viewer; Chromium's tracker for Chromium viewer; and Mozilla's tracker for Firefox viewer). (2) We include bug trackers that contain software that either generates or handles PDF files (GNOME tracker contains Documents [171]; GhostScript tracker handles PDF files [78]; KDE tracker contains Okular [172]; Apache tracker contains FOP [173]; Freedesktop tracker contains LibreOffice [174], pdftohtml [175], and Poppler [77]).

Our scraper supports three tracking systems including Bugzilla [176], Monorail [177], and LaunchPad [178]. For Bugzilla, we searched for the keyword 'pdf' and required an attachment of type `pdf`, `zip` or `tar`. For LaunchPad, since it does not support searching for bugs reports that contain attachments, we simply searched for the keyword 'pdf.' For Monorail, since it does not support the search for specific attachment types, we searched

for bug reports that contain at least one attachment file. The scraper downloads all the PDF attachments from the search results. If the result contains `zip` or `tar` compressed files, it decompresses the files and searches for PDF files.

Fuzzed Corrupted PDF Files Previous work by Force Open [27] released a dataset that contains corrupted PDF files that are generated using fuzzing. It contains 1,723 fuzzed corrupted PDF files containing research papers and PDF books with randomly modified bytes.

4.4.2 Identifying Corrupted PDF Files

The empirical study requires an automated tool to identify corrupted PDF files automatically that are crawled from the bug tracker dataset because (1) there are too many crawled files for manual inspection (6,903 in Table 4.2), and (2) not all the collected files are corrupted. Although there existing tools such as Acrobat Pro DC [179] that can validate PDF files, (1) it is a proprietary GUI tool that does not support automation to scale towards validating thousands of files, (2) previous work [168] had shown that Acrobat Reader contains bugs on the implementation of the PDF specification. Therefore, the automated tool uses seven popular PDF viewers to decide if a PDF file is corrupted by opening the files automatically. Figure 4.3 shows an overview of the empirical study.

The tool first filters out files that are not corrupted because not all crawled files from the bug trackers are corrupted (SanityCheck in Figure 4.3). The tool (1) removes password protected PDF files by detecting them using Mutool's info utility [73], (2) removes any files that are smaller than 300 bytes (we chose a lower threshold to be conservative since a minimal "Hello World" PDF file contains 739 bytes [180]), and (3) removes files that do not start with the magic number, `%PDF`, in the header.

We identify corrupted PDF files by applying screenshot analysis on PDF viewers (SanityCheck in Figure 4.3). A PDF file is considered corrupted if more than one PDF viewer fails to open the PDF file. We select this criterion since previous work had shown cases where one viewer contains a bug [168]. A PDF viewer is considered to have failed to open the PDF file if it displays a blank screen based on the histogram analysis, where the screen only contains pixels of a single color. The primary challenge in the screenshot analysis is that corrupted files often cause unexpected behavior on the PDF viewer and operating system (e.g., crashes and error messages). We utilize optical character recognition (OCR) to detect and close on-screen messages (e.g., informational and error messages) before the screenshot analysis because they interfere against the histogram analysis at identifying

Table 4.1: List of evaluated repair tools and PDF viewers.

Repair Tool	**Version**	**Build Date**	**PDF Library**
mutool [73]	1.12.0	December 2017	Built-in
PDFtk [74]	2.02-4	July 2013	Built-in
GhostScript [181]	9.18	September 2015	Built-in
PDF Viewer	**Version**	**Build Date**	**PDF Library**
Evince [182]	3.18.2	July 2017	GhostScript
Xpdf [183]	3.04	May 2014	Poppler [77]
MuPDF [73]	1.12.0	December 2017	Built-in
Chromium [184]	65.0.3325.18	March 2018	PDFium [185]
Firefox [186]	59.0.2	Match 2018	PDF.js [187]
Acrobat Reader DC [188]	2018.011 (Win. 10)	2018	Built-in
Qpdfview [189]	0.4.17	December 2016	Poppler [77]

content on the screen. The reason to utilize OCR for detecting error messages as opposed to image comparison is that OCR is more reliable at detecting error messages from the user interface. For example, image comparison would fail to recognize a message box if the error message text is different between PDF files. OCR also allows us to adapt to different screen resolutions quickly if needed.

4.4.3 PDF Repair Tools and Viewers

Since the goal of the study is to understand the impact of corrupted PDF files and the repair capability of existing repair tools, we include three common PDF repair tools (Mutool, PDFtk, and GhostScript) and seven PDF viewers (Table 4.1). All software executes on the latest available version on two computers installed with Ubuntu 16.04 LTS and Windows 10 (Acrobat Reader DC). Both computers contain an Intel i5-2400 CPU and 6GB of RAM.

4.4.4 Empirical Study Findings

RQ1: *How many real-world corrupted PDF files exist and what is their impact?*

Procedure (RQ1): Corrupted PDF files can have a negative impact on the end user such as program crashes and security vulnerabilities. Therefore, we want to understand

(1) how many real-world corrupted files exist, (2) if the real-world corrupted files cause severe consequences including security vulnerabilities, and (3) if the real-world corrupted files have an impact to the end user and target software.

To locate real-world corrupted PDF files, we apply screenshot analysis (Section 4.4.2) over the 6,903 crawled files from existing bug trackers (Table 4.2) and automatically identified a list of corrupted PDF files. Since the automated tool is not perfect, we manually inspected the 396 corrupted files. Our manual inspection identified and removed 77 false positives, resulting in a total of 319 corrupted files. There are two primary sources of false positive. The two types of false positive includes (1) the case where the tool failed to take a screenshot using macros, and (2) the crawled file does not contain any content (e.g., the PDF contains no text or images which is impossible to detect). We perform three classifications over the 319 corrupted files (excludes the false positives) to answer RQ1. The first two labels are done independently by two graduate students, and the disagreements are discussed until they are resolved, and the third label is done by a graduate student.

The first classification labels PDF files as either real-world corrupted or purposely corrupted. The goal is to obtain the first set of real-world corrupted PDF files for further studying in RQ3 of this work. For example, a PDF file that is modified manually by hand or modified automatically by fuzzing tools to expose a bug in a viewer would be considered a purposely corrupted file, but a PDF file that is generated by a real-world application (e.g., Microsoft Word) but later corrupted when sent as an email attachment is considered a real-world corrupted file. The labeling process involves manually inspecting the bug report and the corrupted file.

The second classification labels if the PDF file triggers security vulnerabilities in the target software such as PDF applications. For example, a PDF file that triggers an integer overflow vulnerability in a specific PDF viewer or library is labeled as a file that tiggers security vulnerabilities. The labeling process involves manually checking if the bug report is tagged as a security bug and if the developer comments indicate that the PDF file contains security vulnerabilities.

The third classification labels the types of impact that are caused by the corrupted PDF file to the end user and target software. The classification is based on the labels from a previous work that studied bug characteristics [1]. The label list includes program hang, program crashes, data corruption, performance degradation, incorrect functionality, and others. The labeling process involves manually inspecting the bug report and the corrupted file.

Results (RQ1): Table 4.2 shows the number of real-world corrupted PDF files and their classifications. We found a total of 319 corrupted PDF files where 119 of which (37.9%)

Table 4.2: RQ1—A breakdown of the number of PDF files that are crawled from the bug trackers ('crawled'), the number of remaining files after applying SanityCheck ('Filtered'), the number of automatically detected corrupted files ('Corr'), the number of manually identified real-world corrupted files ('Real'), and the number of manually identified files that cause security vulnerabilities ('Sec').

Bug Repo.	**Crawled**	**Filtered**	**Corr.**	**Real.**	**Sec.**
GNOME	439	437	21	18	2
Mozilla	157	157	2	2	0
hromium	2,306	2,276	205	25	117
LaunchPad	2,411	2,374	52	48	0
GhostScript	63	62	6	5	0
KDE	332	323	2	2	0
Apache	256	256	0	0	0
Freedesktop	939	930	31	19	0
Total	6,903	6,815	**319**	**119**	**119**

are real-world corrupted files, while the remaining 198 are purposely corrupted files that causes PDF applications to fail. Real-world corrupted files often come from bad PDF file generators such as cups-pdf [190] that generate a file that does not conform to the specification. The results suggest that there is a significant amount of real-world corrupted PDF files whose content is vital to the user, which motivates the need to repair the file content. Also, these files have not been systematically studied before, which motivates the need to study them.

Amongst these 319 corrupted files, 119 files (30.8%) expose security vulnerabilities in PDF application. It is still important to repair files with vulnerabilities because it can cause severe consequences to the user and help developers localize faults (Section 4.1). And amongst the 119 corrupted files that can trigger security vulnerabilities, 116 of which (99.1%) are purposely corrupted files. The one real-world corrupted file that triggers a security vulnerability triggers an out-of-bounds write error in Chromium's built-in PDF reader (bug #124182 [191]). The bug report contains a reference in the security vulnerability database (CVE-2011-3097), which is confirmed to be a security bug and fixed by the developers.

The classification on the impact of corrupted PDF files is shown in Table 4.3. Corrupted files can commonly cause incorrect functionality on the target software that is unexpected

Table 4.3: RQ1—Classification of the impact of all 119 real-world corrupted PDF files on the end user and target software based on previous work's classification labels [1].

Incorrect Functionality	**Crash**	**Performance Degradation**	**Hang**
94	16	6	3

to the user (94 instances). For example, the target software may not be able to display the content of the corrupted file or have issues at processing the corrupted file. Other common issues include software crashes (18 instances), performance degradation (6 instances), and software hangs (3 instances).

Findings (RQ1): We identified a total of 119 real-world corrupted files from the 319 corrupted files, and identified 119 files that triggers security vulnerabilities from the 319 corrupted files. The PDF file corruption problem is a real problem that often impacts the user with incorrect functionality or even software crashes.

RQ2: *Can existing repair tools repair corrupted PDF files?*

Procedure (RQ2): To check if corrupted PDF files can be repaired by existing tools, we apply existing repair tools (Mutool, PDFtk, and GhostScript) on 1,827 corrupted files (Table 4.2), consisting of both corrupted PDF files from bug trackers (319 files) and fuzzed PDF files from Force Open's fuzzed dataset (1,508 files). We include fuzzed PDF files in the evaluation because it simulates the case of file system errors in the real world, which can corrupt raw bytes of the file. The 1,508 files from Force Open's fuzzed dataset are obtained by applying CorruptCheck on the Force Open dataset of 1,723 files. For RQ2, we study both real-world corrupted PDF files and purposely corrupted PDF files because it is beneficial to repair both types as discussed in the introduction. We apply existing repair tools to repair the corrupted files, and then verify if the corrupted files had been repaired successfully by the repair tools (Figure 4.3).

Results (RQ2): Table 4.4 presents the repair result: 1,267 of the 1,827 corrupted files cannot be repaired by any existing repair tools (Mutool, PDFtk, and GhostScript). Specifically, 231 of the 319 corrupted real-world files cannot be repaired by any existing repair tools, while 1,036 of the 1,508 corrupted fuzzed files cannot be repaired by any existing repair tools.

As discussed in RQ1, the 319 corrupted files contain 119 real-world corrupted files and 119 files that cause security vulnerabilities respectively. Table 4.4 shows that amongst the

Table 4.4: RQ2—Number of corrupted files ('Corr.') that existing tools (Mutool, PDFtk, and GhostScript) cannot repair ('None rep.'). We also show the number of files, amongst the ones that nobody can repair, that are real-world corrupted ('None Rep. Real.') and triggers security vulnerabilities in the target software ('None Rep. Sec.').

Set	**Source**	**Corr.**	**None Rep.**	**None Rep. Real.**	**None Rep. Sec.**
1	GNOME	21	12	9	2
	Mozilla	2	2	2	0
	Chromium	205	162	11	103
	LaunchPad	52	32	29	0
	GhostScript	6	5	4	0
	KDE	2	0	0	0
	Apache	0	0	0	0
	Freedesktop	31	18	8	0
	Set 1 Total	319	231	63	105
2	Force Open	1,508	1,036	0	n/a
	TOTAL	**1,827**	**1,267**	**63**	**105**

119 real-world corrupted files, 63 of which cannot be repaired by existing tools; and from the 119 files that cause security vulnerabilities, 105 of which cannot be repaired by existing tools.

Findings (RQ2): There is a need for a better repair technique since 1,267 of the 1,827 corrupted files cannot be repaired by any existing repair tools (Mutool, PDFtk, and GhostScript).

RQ3: *What types of real-world file corruption exist?*

Procedure (RQ3): To study the impact of file corruption, we perform a classification over the 119 real-world corrupted files to discover the causes of corruption. The classification requires inspecting the raw bytes of a PDF file, the file's associated bug report, and the output of PDF viewers. The initial step discovers the full list of topics (causes of corruption), which is used for open card sorting [192]. Open card sorting is a technique for organizing items into categories. The purpose is to identify high-level causes of corruption from detailed corruption causes. We follow the open sorting process digitally using xSort [193]. We first enter a unique list of corruption causes into the xSort software for

Table 4.5: RQ3—Causes of file corruption (Type Coor.) on the 119 real-world corrupted files. The table shows the number of files for each cause of corruption (T.); the number of files that that cannot be repaired by existing repair tools including Mutool, PDFtk, and GhostScript (F.); the number of files that can be repaired by the proposed technique, DocRepair, which is described in Section 4.5 (R.); and the corresponding repair operator that is implemented in DocRepair to address the different causes of corruption (RO).

Type Corr.	**T.**	**F.**	**R.**	**RO**	**Type Corr.**	**T.**	**F.**	**R.**	**RO**
Base Structure	14	9	10	RO1, RO3, RO5	Corrupted Data Stream	10	4	0	
Font Resource	9	2	0	RO3	Page Tree Structure	6	3	2	RO4
Bad Data Field Value	5	3	1	RO3–RO6	File Trailer	5	1	3	RO6
Image	4	3	0		Form	4	0	3	
Colorspace	4	1	0		Cross-Reference Table	3	1	1	RO7
File Encryption	3	2	0		Graphics Gradient	3	3	0	
Embedded Javascript	1	1	0		Unknown	48	30	6	

sorting. The presentation of the cards is randomized by the xSort, and we create subcategories during the sorting process. The goal is to sort the topics into categories and assign a name that best describes the content of each category.

A total of 119 topics (causes of corruption) are discovered before the initial classification of the 119 real-world corrupted files. Since multiple corrupted files may have the same topic, we perform string matching to obtain a list of unique topics for the open card sorting process. The unique list of topics are entered into the card sorting software, and the user sorts the topics into categories (Table 4.5).

Results (RQ3): Table 4.5 shows the causes of file corruption from the 119 real-world corrupted files. We explain the identified types of file corruption below.

'Base structure' contains files that are corrupted due to file transfer or file-system bugs, where a large chunk of the PDF file is missing and thus destroying the base structure of the file. For example, the PDF objects (nodes in Figure 4.2) can be missing from the file

```
7 0 obj
<< /Type /Font
/Subtype /TrueType
/BaseFont /DejaVuSans <- invalid font reference
/FirstChar 0
/LastChar 5
/FontDescriptor 6 0 R <- invalid font descriptor reference
/Widths [ 1178 1253 1300 842 0 ]
>>
endobj
```

Listing 4.3: Example of an invalid font object where the embedded font cannot be loaded by PDF viewers [194].

which can cause parsing issues. Therefore, it is important for repair tools to take into consideration the situation where multiple objects are missing during a repair.

'Corrupted data stream' contains files that contain compressed content (e.g., with encoders such as Flate, LZW and ASCIIHex) that had been corrupted, which causes the encoded content to be unrecoverable. However, it may be possible to recover other parts of the PDF file if the corrupted encoded data does not impact too many parts of the file. It may not be advisable to encode all data into a single data stream since a single byte corruption in the encoded data can potentially corrupt the entire data stream.

'Font Resource' contains files with corrupted file resource, which is required to display the text. When a file does not include the needed font embedding in the file, it can cause PDf viewer issues at displaying the content. It is also the reason that other PDF format extensions such as PDF/A specify the requirement to embed all fonts in a PDF file for long-term preservation purposes. An invalid font component of a PDF file generated by GTK+ is shown in Listing 4.3 where the PDF file references a front that is not loadable by Evince and other viewers [194]. The font (`DejaVuSans`) is invalid because the viewer cannot load the embedded font. There are also other cases where a font is not embedded in the PDF file. For example, LibreOffice [174] generated a corrupted PDF file that requests a font that is not embedded in the PDF file [195].

'Page tree structure' contains files with an invalid page tree structure. The page tree structure is responsible for organizing the display of pages within the PDF file. Common real-world corruption scenarios include cases where a part of the page tree of a PDF file is missing causing PDF viewers to fail to open the file [196, 197]. Listing 4.4 contains an example of a corrupted page tree [198], where the page tree structure is missing an

```
1     11 0 obj
2     <</Metadata 2 0 R
3     /PageLabels 6 0 R
4     /Pages 8 0 R <- invalid object reference
5     /Type/Catalog>>
6     endobj
7
8     12 0 obj
9     <</Contents 14 0 R
10    /CropBox[0 0 612 792]
11    /MediaBox[0 0 612 792]
12    /Parent 8 0 R
13    /Resources 19 0 R
14    /Rotate 0
15    /Type/Page>>
16    endobj
```

Listing 4.4: Example of a corrupted page tree structure where the 'Pages' object (object number 8 which is referenced on line 4) is missing [198].

important object (8 0 R) called the 'Pages' object. The 'Pages' object is responsible for letting the PDF viewers know the full list of available displayable pages in the PDF file (e.g., object number 12 at the bottom of the listing).

'Bad data field value' contains files with specific issues in the key-value pairs within an object. For example, Launchpad's Evince PDF viewer fails to print a PDF file due to an invalid value [199] for the key `Encoding` in Listing 4.5. Based on the PDF documentation [166], the value should be a name object that starts with the character '\'. For example, '\WinAnsiEncoding' is a potential valid value.

'File Trailer' contains files with invalid file trailers. An invalid file trailer is shown in Listing 4.1 where it contains a reference to an invalid object number (-406081386). The value should be a valid object number that exists in the PDF file. There are also cases where a file trailer is missing from the file due to bad applications. For example, scanner application Simple Scan [200] generates corrupted PDF files that do not contain a file trailer at all [201, 202].

'Image' contains files with embedded images and graphics that are invalid. The issue can be caused by the corrupted binary pixel data, the color profile of the image data, or the image encoding algorithm. A common issue is that the image parser can fail while parsing the input image stream. For example, Launchpad's Evince PDF viewer crashes

```
1 1 0 obj
2 <</BaseFont /Courier
3 /Subtype /Type1
4 /Name /F1
5 /Type /Font
6 /Encoding WinAnsiEncoding <- invalid value for the key, Encoding
7 >>
```

Listing 4.5: Example of an invalid data field value for the key `Encoding` [199], where the key `Encoding` has an invalid value.

when opening a PDF that contains an embedded image that is compressed using JBIG2 algorithm [203].

'Form' contains files with invalid fillable forms. It can relate to issues such as the inability to save the data entries and bad text characters within the forms.

'Colorspace' contains files with invalid Colorspace, which is a color profile that define the format of the embedded image pixel data [204]. It is important to understand that the PDF file format does not store images as a specific file format (e.g., PNG or JPG), but instead stored images as a binary pixel data that is defined by its color profile called Colorspace. While the 'Image' category contains PDF files with issues with image embedding, the 'Colorspace' category contains PDF files that have issues that are directly caused by the color profile.

'Cross-reference table' refers to a specific part of the PDF file that is used by PDF applications to speed up the loading speed.

'File Encryption' refers to the encryption functionality that protects the content of a PDF file.

'Graphics gradient' refers to a drawing component within the PDF file format, which are shading patterns that defines a smooth color transition between on point to another. The PDF file format defines a list of such shading patterns to allow a wide range of coloring patterns.

'Embedded Javascript' refers to the Javascript code that is embedded within the PDF file which allows for interactive content.

'Unknown' refers to corrupted files that we cannot locate the cause of corruption. The identification of the cause of corruption is a challenging task since bug reports may not document the cause of the file corruption.

Findings (RQ3): The findings on the causes of corruption in Table 4.5 inspired us to implement seven repair operators to address them. Specifically, the proposed repair operators focus the repair over the most frequent causes of corruption including base structure, font resource, page tree structure, bad data field value, file trailer, and cross-reference table.

4.5 Design and Implementation

DocRepair repairs PDF files with two main steps as shown in Algorithm 1. First, it parses the corrupted file and collects data for specific objects in the file (Section 4.5.1). Second, it applies modifications to the file using a list of repair operators in a specific order (Section 4.5.2), where the repair operators are designed based on the identified causes of file corruption from the empirical study in RQ3 (Table 4.5).

4.5.1 Data Parsing and Collection

The first step in the repair algorithm (line 1 of Algorithm 1) parses a PDF file and collects data from specific objects in the file. The collected data is used in later stages by the repair operators (Section 4.5.2).

The function `parse()` in line 1 of Algorithm 1 collects information needed for detecting and repairing potential bugs in the PDF file (line 1). Its purpose is to collect structural information of important objects (i.e., trailer, catalog, page tree) and the relationship between these objects (i.e., parent and child relationship). The function also performs validation over the parsed data to ensure they are semantically correct (i.e., ensure objects and references are valid).

While several PDF parsing libraries exist, we re-implemented a parser (adapted from Mutool's PDF parsing library [73]) to overcome limitations of existing libraries. Since parsing libraries are designed to parse correctly formed PDF files, existing parsers would stop the parsing and return an error if the PDF file is too corrupted. To overcome this issue, DocRepair parses one object at a time (an object can be a boolean, a number, a dictionary, a data stream, etc.) to confine the propagation of parsing errors.

4.5.2 Repair Operators

The second step of the repair algorithm (line 2—20 of Algorithm 1) contains a list of repair operators to repair the PDF file. The repair operators analyzes the collected data to flag

Algorithm 1: Algorithm for repairing PDF files. RO1–RO7 denotes the seven repair operators.

Input : input PDF file buffer - *inputFile*
Output : repaired PDF file buffer - *parsedPDF*

$parsedPDF \leftarrow parse(inputFile)$
`/* RO1 - Object spacing */`
$parsedPDF \leftarrow removeBadSpacing(parsedPDF)$
`/* RO2 - Header */`
$parsedPDF \leftarrow repairHeader(parsedPDF)$
`/* RO3* - Font Resource */`
for $font \in parsedPDF.fontReferences$ **do**
  if $!existObject(parsedPDF, font)$ **then**
    $parsedPDF \leftarrow parsedPDF + genTemplate(\text{“}font\text{”}, font.id)$
  end
end
`/* RO4* - Page Tree Structure */`
if $parsedPDF.pageTree.repair$ **then**
  $parsedPDF \leftarrow parsedPDF + genTemplate(\text{“}page\text{”}, parsedPDF.pageTree.id)$
  $parsedPDF \leftarrow updateReference(parsedPDF, \text{“}page\text{”})$
end
`/* RO5* - Catalog */`
`/* Check if catalog is missing or the child (page node) is missing */`
if $parsedPDF.catalog.id = \emptyset$ **or** $parsedPDF.trailer.child = \emptyset$ **then**
  $parsedPDF \leftarrow parsedPDF + genTemplate(\text{`}catalog\text{'}, parsedPDF.catalog.id)$
  $parsedPDF \leftarrow updateReference(parsedPDF, \text{“}catalog\text{”})$
end
`/* RO6 - Trailer */`
$parsedPDF \leftarrow remove(parsedPDF, \text{“}trailer\text{”})$
$parsedPDF \leftarrow parsedPDF + genTemplate(\text{“}trailer\text{”}, parsedPDF.catalog.id)$
`/* RO7 - xRef table */`
$parsedPDF \leftarrow remove(parsedPDF, \text{“}xRef\text{”})$
$parsedPDF \leftarrow parsedPDF + genTemplate(\text{“}xRef\text{”}, parsedPDF)$
return $parsedPDF$

Table 4.6: A list of proposed and existing repair operators (RO) and whether existing repair tools support them (* means the repair operator is unique to DocRepair). 'Repaired' shows the usage frequency of each repair operator over the 354 corrupted files that are repaired by DocRepair. 'Uniquely Repaired' shows the usage frequency over the 10 corrupted files that are uniquely repaired by DocRepair.

Repair Tool	**RO1, Object Spacing**	**RO2, Header**	**RO3*, Font Resource**	**RO4*, Page Tree Structure**	**RO5*, Catalog Object**	**RO6, File Trailer**	**RO7, Reference Table**
MuPDF	Yes	Yes	No	No	No	Yes	Yes
PDFtk	Yes	Yes	No	No	No	No	Yes
GhostScript	Yes	Yes	No	No	No	No	Yes
DocRepair	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Repaired	92	5	12	198	218	354	354
Uniquely Repaired	6	2	5	9	9	10	10

bugs in the PDF file, and modifies the PDF file to address the flagged bugs. DocRepair includes both (1) newly proposed repair operators and (2) repair operators from existing repair tools. Table 4.6 shows a summary of the designed and implemented repair operators.

Amongst the seven repair operators, DocRepair contains three that no existing tools have, which are designed based on the findings from our empirical study in Section 4.4.4. Table 4.5 shows the mapping between the causes of corruption against the implemented repair operators in column 'RO.' We do not propose repair operators for two of the causes of corruption ('Corrupted data stream' and 'Image'), and we discuss the reasoning at the end of this section.

The main causes of corruption are covered by different repair operators (RO1–RO7) in Table 4.6. A repair operator may address multiple causes of file corruption. For example, when the base structure of a PDF file is corrupted (a cause of corruption) it may affect both the font and header of the file, which means it would require two repair operators to fully repair the file.

The mapping between the cause of corruption and the corresponding repair operator is discussed below. 'Base Structure' is covered by RO1 (object spacing), RO3 (font resource), and RO5 (catalog object). 'Font Resource' is being addressed by RO3 (font resource).

'Page Tree Structure' is addressed by RO4 (page tree structure). 'Bad data field value' is addressed by applying multiple repair operators (RO3—RO6) through detecting for invalid object number or references. 'File Trailer' is directly addressed by RO6 (file trailer). 'Reference table' is directly addressed by RO7 (cross-reference table). We do not propose repair operators for the remaining causes of corruption that are in Table 4.6 due to their low impact.

The seven implemented repair operators (RO) are annotated as a comment on the right of the algorithm (Algorithm 1). Five of the seven proposed repair operators invoke the function `genTemplate()` which generates a new object using templates for components such as the cross-reference table and font object. The function `genTemplate()` generates the repair with the name of the object passed as the argument of the function.

We describe the seven repair operators below. Three of the repair operators are new and have not been used in previous document repair tools. Repair operators that have not been implemented by existing repair techniques (RO3, RO4, and RO5) are annotated with a star symbol (e.g., RO3*).

RO1: Object Spacing

Line 2 in Algorithm 1 checks the spacing between PDF objects and ensures that it only contains valid spacing characters between PDF objects, as defined in the file format specification [166]. RO1 is performed first since a corrupted file may contain invalid characters that can lead to parsing failures to stop the rendering of PDF files.

RO2: Header

Line 3 checks if the file header is correct because PDF applications may utilize the header to determine the file type. RO2 ensures that the magic number is correct, and the bytes after the magic number contain only 'high bit' ASCII characters of value 128 or higher. The ASCII characters can be used by file transfer applications to determine if the file's content contains binary or text data.

RO3*: Font Resource

Repairing font objects is important because a missing or corrupted font object often causes the text that is within the document to be undisplayable, even if the text is embedded in the file and not corrupted.

```
1 FONT_ID 0 obj
2 << /Type /Font /Subtype /Type1
3 /BaseFont /FONT_NAME>>
4 endobj
```

Listing 4.6: Template for the font object.

Lines 4 to 8 of Algorithm 1 iterate through font resources and detect any missing or corrupted fonts. If any of the font resources are missing or corrupted, RO3 invokes `genTemplate()` to generate a new font object to replace the missing resources.

Listing 4.6 shows the template for generating a font object, which is a dictionary object that consists of key-value pairs (i.e., the key 'Type' contains the value 'Font'). In this example, the template's object number `FONT_ID` has to be substituted with the correct object number. The variable `FONT_NAME` has to be substituted with the correct name of the missing font. These two values are extracted during the parsing step in Section 4.5.1.

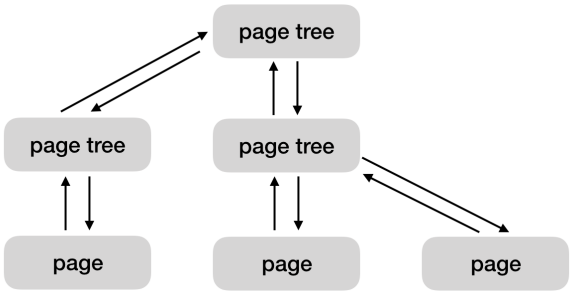
RO4*: Page Tree Structure

The page tree structure represents the page layout of a PDF file. The page tree hierarchy is shown in Figure 4.4. A 'page tree' is responsible for organizing 'page' objects, where a 'page' object defines the content of a single page. Both 'page tree' and 'page' objects contain pointers (arrows in Figure 4.4) that connect objects. RO4 utilizes the parent-child pointer relationship between these objects that are extracted during the parsing process (line 1).

Lines 9 to 12 fix the page layout of a PDF file. Two pieces of information are necessary for this repair: the object ID of the top-level 'page tree' object and a list of 'page' object references. The top-level 'page tree' object is located by tracing through the pointers until it can locate the top-level 'page tree' object (root node in Figure 4.4). The list of 'page' object references is obtained by checking the type of the objects (leaf nodes in Figure 4.4).

If the page tree structure is corrupted (there may be multiple page trees within a PDF file), RO4 generates a new page tree template (line 10) as the new root node. Although a PDF file may contain multiple page tree objects, the proposed repair operator only generates a single page tree object. The new page tree object aggregates all the available 'page' objects that are responsible for holding the actual page content under the new top-level 'page tree' object. RO4 also updates the pointer references of the page objects since a new 'page tree' object is generated (line 11).

Figure 4.4: Hierarchy of the 'Page Tree' structure. The tree contains intermediate nodes called 'page tree' and leaf nodes called 'page object'. The arrows represent an indirect reference between the objects.




```
1 PAGE_TREE_ID 0 obj
2 <</Type /Pages /Kids PAGE_OBJECT_LIST
3 /Count PAGE_OBJECT_LIST.LENGTH>>
4 endobj
```

Listing 4.7: Template for the page tree object.

An example is shown in Listing 4.7, where `PAGE_TREE_ID` is the new top-level 'page tree' object, and `PAGE_OBJECT_LIST` is the referenced list of 'page' objects that represents the content of the file.

Although RO4 will repair the page tree structure, the current implementation does not analyze the order of the pages, and hence the final PDF file may contain randomized page order. The repair operator is unique to DocRepair, and it is essential for recovering displayable content. Amongst the 354 successfully repaired PDF files by DocRepair, 198 of which utilized this repair operator (Table 4.7).

RO5*: Catalog Object

The catalog object represents the root of the PDF tree structure (Figure 4.2), which contains references to other core parts of the PDF file. Lines 13–16 fix the document catalog object. It performs a repair if a catalog object cannot be found or if the catalog object cannot be located from the file trailer. The generated template (line 14) is shown in Listing 4.8, where `CATALOG_ID` and `PAGE_TREE_ID` have to be inferred similarly to RO4. RO5 has to be applied after the RO4 because RO5 uses the repaired `PAGE_TREE_ID` from RO4

```
1 CATALOG_ID 0 obj
2 <</Type /Catalog /Pages PAGE_TREE_ID 0 R>>
3 endobj
```

Listing 4.8: Template for the catalog object.

```
1 CATALOG_ID 0 obj
2 <</Root CATALOG_ID 0 R /Size TOTAL_NUM_OBJECTS>>
3 endobj
```

Listing 4.9: Template for the trailer object.

if the page tree structure is corrupted. Similar to RO4, RO5 invokes `updateReference()` to update the external reference from the trailer object (line 15).

RO6: Trailer Object

The trailer object is the entry reading point for applications that process PDF files. Lines 17 to 18 fix the trailer object. Existing techniques only regenerate the trailer object if the trailer object cannot be found or cannot be parsed correctly in the PDF file, whereas DocRepair validates the correctness of the trailer data and triggers a repair upon detecting invalid values. For example, the trailer object requires a reference to the catalog object, but existing tools do not check if the existing reference actually points to a catalog object.

The object template is shown in Listing 4.9. RO6 has to be applied at the end of the repair because the previous repair operators (RO3 and RO4) modify the total number of objects within a PDF file (`TOTAL_NUM_OBJECTS`). In addition, RO6 uses the variable `CATALOG_ID` from RO5.

RO7: Cross-reference Table (xRef)

The cross-reference table stores a pointer reference to each PDF object. The cross-reference table is often used by PDF readers to fetch information quickly without reading the entire PDF file. Lines 19 to 20 reconstruct the cross-reference table of the PDF file. RO7 is a common repair operator that is deployed by all PDF viewers and repair tools.

Excluded Categories

We do not include a repair operator for two causes of file corruption, 'Corrupted data stream' and 'Image,' for three reasons. The decision is made based on a manual inspection of all the corrupted files that cannot be repaired by any existing tool in Table 4.5.

The first reason is that the files are too corrupted and are beyond repair. The six files in the 'Corrupted data stream' category that can be repaired by existing tools have a corrupted data stream that only affects a minor part of the PDF file. However, the four files that cannot be repaired by existing tools contains too many corrupted components. For example, two of the files from Chromium bug #444446 are damaged beyond repair, with hundreds of corrupted objects. In another example, one of the files from Freedesktop bug #14098 is created by a 2D graphics library with incorrectly embedded images, and since the only content within the file is an image and the image data is corrupted, it is impossible to recover any content from the file.

The second reason is that the files require the design of a repair operator that is too specific to the file and are not generalizable. For example, Freedesktop bug #6688 contains a file that is corrupted with a unique pattern that is unlikely to occur in another corrupted file. While this file could be repaired, the RO used to repair this file would be very unlikely to generalize to other corrupted files.

Third, the files in the 'Image' category require the support for image analysis techniques which is beyond the scope of this work. For example, a corrupted image data within a PDF file can affect the binary pixel data, the color profile of the image data (also known as the colorspace which has its own category), or image encoding algorithm (also known as filters). Since there are many color profile configurations and ten different encoding algorithms supported by the PDF file format, we do not propose new repair operators to address these files. For example, Freedesktop bug #6500 contains an image that is encoded using the JBIG2 filter, where the decoded data contains a byte that causes the PDF rendering library, Poppler, to fail to render the image since Poppler's implementation does not expect the extra NULL byte in the binary data. Since the repair requires a change to the JBIG2 filter's algorithm, we do not implement a repair operator for this file.

4.6 Evaluation Method and Results

RQ4: *How does DocRepair compare to existing file recovery tools?*

Table 4.7: RQ4—A summary of the automated screenshot evaluation results over 319 corrupted files (including real-world corrupted and purposely corrupted files) and 1,508 fuzzed corrupted files. The table shows the number of repaired files by each specific repair tool (Mutool, PDFtk, GS and DocRepair). The number in bracket refers to the number of files that are uniquely repaired. 'All' includes the number of real-world corrupted and purposely corrupted files. 'RC' only includes real-world corrupted files.

Source	Num. Corr. All	Num. Corr. RC	Mutool All	Mutool RC	PDFtk All	PDFtk RC	GS All	GS RC	Doc All	Doc RC	Doc+ All	Doc+ RC
GNOME	21	18	8 (0)	8 (0)	1 (0)	1 (0)	9 (1)	9 (1)	4 (1)	4 (1)	10	10
Mozilla	2	2	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0	0
Chromium	205	25	17 (5)	9 (1)	20 (3)	6 (0)	29 (13)	10 (3)	18 (6)	11 (3)	49	17
LaunchPad	52	48	14 (1)	13 (1)	4 (0)	4 (0)	14 (5)	13 (5)	9 (2)	9 (2)	22	21
GhostScript	6	5	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	0 (0)	0 (0)	1	1
KDE	2	2	2 (0)	2 (0)	0 (0)	2 (0)	2 (0)	2 (0)	1 (0)	1 (0)	2	2
Apache	0	0	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0	0
Freedesktop	31	19	7 (2)	5 (1)	3 (0)	2 (0)	11 (6)	10 (6)	6 (1)	5 (1)	14	12
	319	119	49 (8)	38 (3)	29 (3)	16 (0)	66 (25)	45 (15)	38 (10)	30 (7)	98	63
Force Open	1,508	none	459 (347)	none	12 (0)	none	18 (3)	none	316 (201)	none	673	none
Total	1,827		508 (355)		41 (3)		84 (28)		354 (211)		751	

Procedure (RQ4): To compare our proposed technique, DocRepair, against existing file recovery tools (Mutool, PDFtk, and Ghostscript), we perform an evaluation over both real-world and fuzzed corrupted PDF files that are identified in the empirical study (Table 4.2).

Each PDF repair tool is executed on the corrupted PDF file to generate a new PDF file that represents the potentially repaired version. To verify if the repaired file is indeed successfully repaired, we utilize CorruptCheck to check if the repair is successful (Section 4.4.2).

In addition, we propose and evaluate a technique called DocRepair+ that evaluates the result of multiple repair tools and returns the PDF file(s) that represents the best repair amongst a list of repair candidates. To measure the quality of a repair, we use a simple metric that determines the quality of a repaired file based on the total number of PDF viewers that can successfully open and display the file's content. DocRepair+ measures the repair quality of the four repair tools' generated PDF file by locating the repaired candidate(s) that contains the lowest number of failure amongst seven different PDF viewers.

Results (RQ4): The result of the comparison against existing repair tools' repair capability is shown in Table 4.7. For the real-world corrupted and purposely corrupted files ('All' column in Table 4.7), amongst the 1,827 corrupted files from the two corpora, DocRepair is capable of repairing 354 files while Mutool, PDFtk, and GhostScript repaired 508, 41 and 84 respectively. The number in brackets shows the number of uniquely repaired files

by each tool. Although Mutool uniquely repaired 355 in total, DocRepair can complement existing repair tools by repairing 211 additional files that none of the existing repair tools can repair.

We also show the data for real-world corrupted files ('RC' column in Table 4.7). DocRepair repaired a total of 30 real-world corrupted files where 7 of which cannot be repaired by any existing repair tools. Mutool, PDFtk, and GhostScript repaired 38, 16 and 45 real-world corrupted files respectively.

To understand each repair operator's contribution towards a successful repair, we measured the repair operator's usage frequency over the 354 corrupted PDF files that had been successfully repaired by DocRepair (Table 4.6). The data shows that RO6 (file trailer) and RO7 (cross-reference Table) are always applied during a repair. RO1 (object spacing), RO4 (page tree structure) and RO5 (catalog object) are also frequency used. RO2 (header) and RO3 (font resource) are infrequently used since it is less common for these two components to be corrupted. A median of four operators are used to repair the 354 corrupted files.

We show an example of a PDF file that requires the application of multiple repair operators. Chromium bug #70440 contains a PDF file that is corrupted when it is saved using the Chrome browser [205], where six of the seven repair operators are utilized to repair the file. A diff of the changes are shown in Listing 4.10. RO1 (object spacing) is applied because the file contains trailing garbage at the end of the file. RO3 (font resource) is applied because the referenced font object, `T1_2`, is missing from the file, so it generated a temporary font as a substitution. RO4 (page tree structure) and RO5 (catalog object) are applied because the 'page tree' object and catalog object are both missing due to file corruption. RO6 (file trailer) and RO7 (reference table) are always applied during a file repair.

The repair capability of the combined repair technique, DocRepair+, is shown in the last column of Table 4.7. DocRepair+ repaired 751 corrupted files, which is far more than any individual tool is capable of repairing. It shows that the proposed metric is effective at determining the quality of the file repair.

Findings (RQ4): The result shows that the proposed technique, DocRepair, is effective at repairing corrupted PDF files (contributing 211 unique repairs among the 1,827 corrupted files). The proposed combined technique, DocRepair+, which relies on screenshot analysis to determine the best repair is also shown to be effective (repairing 751 of the 1,827 corrupted files). The proposed repair operators are capable of contributing to unique file repairs (e.g., RO4 helped DocRepair repair 218 of the 354 corrupted files and a median of four repair operators is used to repair corrupted files).

```
// removed garbage data using RO1

// RO3
80 0 obj
<< /Type /Font /Subtype /Type1 /BaseFont
/Times-Roman >>
endobj

// RO4
82 0 obj
<</Type /Pages /Kids [17 0 R ]
/Count 1>>
endobj

// RO5
83 0 obj
<</Type /Catalog /Pages 82 0 R>>
endobj

// RO7
xref
0 5
0000000000 65535 f
0000021329 00000 n
0000021733 00000 n
0000021867 00000 n
0000022595 00000 n
15 1
0000000016 00000 n
17 16
0000000477 00000 n
0000000936 00000 n
...

// RO6
trailer
<</Root 83 0 R /Size 84>>
startxref
23361
%%EOF
```

Listing 4.10: An example of DocRepair applying multiple repair operators on the corrupted PDF file in Chromium bug #70440.

4.7 Threats to Validity

We discuss the different threats to validity to our experiments.

4.7.1 Internal Validity

Internal validity concerns the extent where one can claim the cause and effect between variables in the experimental design.

The empirical study may contain external variables that can influence the authors' analysis of the PDF files. A person's experience with the PDF file format may impact the classification of the different causes and impact of corrupted files. We mitigate the effect by having multiple authors to perform the analysis for parts of the classifications that heavily depends on the knowledge of the inspector.

4.7.2 External Validity

External validity concerns if the conclusions can be generalized outside of the study.

The empirical study currently only contains files that are crawled from existing bug trackers which can cause a selection bias towards files that are corrupted due to specific software. For example, files that are corrupted due to file system bugs or network transfer bugs (ones that are not associated with a specific software application) may be less represented in the dataset. To mitigate the bias one may collect PDF files directly from existing web pages instead of only focusing on bug trackers. However, the collection of PDF files from existing web pages can be a very challenging task.

The detection of corrupted PDF files is done using seven specific PDF viewers including Evince, Xpdf, MuPDF, Chromium, Firefox, Adobe Reader DC, and Qpdfview. There are other popular PDF viewers from Windows and Linux that may be used to help determine if a file is corrupted. The current selection of PDF viewers consists of mostly Linux PDF viewers which are a potential bias. However, some of the PDF viewers such as MuPDF, Chromium, and Firefox are cross-platform which mitigates the threat.

The evaluation of the proposed technique is only being evaluated against three existing PDF repair tools including Mutool, PDFtk, and GhostScript. While these are the most popular PDF repair tools, there may be other better repair tools to further expand the generality of the conclusions.

4.7.3 Construct Validity

Construct validity concerns if a test measures the intended construct of the study.

The effectiveness of a PDF repair tool is measured based on the number of PDF viewers that can successfully open and display the content of the corrupted file after the repair. It might not be the best way to measure the effectiveness of a repair tool as it does not measure the amount of content that is recovered by the proposed repair technique. Previous work from Force Open proposed to utilize Levenshtein distance to measure the amount of content that can be recovered by the PDF repair tools [27]. However, their evaluation result shows that the corrupted files from their dataset are either completely repaired or not at all. We analyzed Force Open's dataset and is able to attribute this to the approach that they create their fuzzy dataset, where a file is only considered to be corrupted if it cannot be opened by a PDF processing tool called *pdftotext*.

4.7.4 Conclusion Validity

Conclusion validity concerns the if the reached conclusions about the relationships in the data are reasonable.

The empirical study relies on the observer's knowledge on the PDF file format. Although the observer who analyzed the corrupted files had a year of experience working with the PDF file format and had spent much time to understand the corrupted files, the conclusions about the corrupted PDF files may still not be accurate because the PDF file format is complex.

The analysis in the empirical study relies on studying the bug reports for the corrupted PDF files. Since each corrupted PDF can be mapped to a bug report, it helps the observer to analyze the corrupted PDF files. However, bug reports may contain inaccurate information about the corrupted PDF file.

Another threat concerns the ability of an observer to determine if a corrupted file is fully recovered. The reason is that the observer does not have a non-corrupted version of the file. The observer can only judge the correctness of the file structure based on manual inspection of the raw data, the bug report associated with the corrupted file, and the output from PDF viewers and recovery tools.

4.8 Summary

Data corruption is one of the most common issues that impact the availability of user's data and causes software failure. This work proposed the first comprehensive dataset of 119 real-world corrupted files. We conducted an empirical study to understand the number of real-world corrupted files in the wild, the causes causes and impact of file corruption, and the repair capability of existing repair tools. The study findings motivated the design of three unique repair operators to help repair corrupted files. We proposed DocRepair and performed an evaluation against three common PDF recovery tools including Mutool [73], PDFtk [74] and GhostScript [78]. We also proposed DocRepair+ which repairs corrupted files through the combination of multiple repair tools. The result shows that DocRepair is capable of complementing existing repair tools by repairing 211 additional corrupted files that none of the existing repair tools can repair.

Chapter 5

Automatic Documentation Generation: Crowd Sourced Comment Generation

5.1 Motivation

Code commenting has been an integral part of software development and standard practice in the industry. Code comments improve software maintainability [83] and programming productivity [4] by helping developers understand the source code. Also, they improve software reliability through assisting in detecting software defects [8]. Despite the need and importance of commenting code, many code bases are not commented or not adequately commented [2].

Therefore, it would be beneficial to generate comments automatically when possible to improve software maintainability and reliability. Also, since it is time-consuming for developers to write comments, automatic comment generation can save developers' time in writing comments so that developers can spend their valuable time on other tasks, which helps to keep the source code up-to-date against the code comments. Previous work had shown that newly added code barely gets commented and that when code gets commentated it is mostly done on class and method declarations instead of method calls [206]. Therefore, an automatic comment generation technique at the code fragment level, as opposed to class and method level, would be beneficial to the developers.

Researchers have proposed techniques to generate comments from source code automatically. Sridhara et al. automatically generate a summary comment for a Java method [89]. They leverage the code structure and linguistic information from identifiers using the Soft-

ware Word Usage Model (SWUM) [92], and generates one comment sentence for each chosen statement from the method. After that, they concatenate the comment sentences to form a summary comment for the method. In a follow-up project, they identify statements with similar structures and topics to generate comments for groups of statements [90]. While these techniques are successful initial steps toward automatic comment generation, they have two main limitations. First, the techniques can only generate comments for particular code structures. For example, they can generate comments for one method body [89], groups of method calls [90], groups of if-else statements [90], method parameters [88], or classes [91]. Second, the performance of their work depends on high-quality identifier names and method signatures. For example, when grouping method calls, it requires that all method names contain the same verb [90]. If the identifiers and methods have poor names, then the approach may fail to generate accurate comments or any comments at all.

Recent state-of-the-art by McMillan et al. proposed to generate documentation that includes context information for Java methods [207]. They collect contextual data about the methods from the source code (e.g., statement which called the method, statements which supplied the method's input, and the method that uses the method's output) and use the keywords from the context of the methods to describe them. Their work labels method signatures using SWUM [92] to generate a natural language sentence to describe the functionality of the method, and extracts the contextual information of the method using PageRank to generate additional sentences as a part of the summary. The differences between their technique and our proposed technique are that their work performs summarization on the method level whereas our work focuses on generating code comment for code segments inside a method, and their summary includes the context of the method. Although their work relies on SWUM, their technique is capable of generating comments for any method since the technique is only dependent on SWUM's ability at identifying the parts-of-speech from the method signature. Our work is less dependent on the quality of the identifier names because it only requires at least one text similarity term between the code comment and the code segment to establish a link, and it utilizes a crowdsourced approach for automatic comment generation where it selects the best comment amongst a range of candidates from Stack Overflow and code sources such as GitHub.

We propose a new approach to generate comments automatically to address the above limitations. We observe that Question and Answer (Q&A) sites such as Stack Overflow [31] naturally contain code descriptions and open-source projects contain code comments written by developers. Therefore, we can extract code comments from the two available sources for automatic comment generation. Specifically, Stack Overflow is widely used to ask questions such as code development and debugging. The questions on Stack Overflow often

receive high-quality answers due to the large user base. For example, a user asked, "*Can I know if a given method exists?*" (Stack Overflow post `#28069121`) to check if a specific method exists within a class. The question received a Java code snippet that checks if a given method had been declared in the class. We can use the statement form of the question "*Know if a given method exists.*" as an explanatory description of the code snippet. We refer to the code snippet and description as a *code-description mapping*. If a code segment in a software project is identical or similar to the above code snippet, then the corresponding description can be an explanatory comment for the code segment in the software project.

Stack Overflow [31] contains a wealth of information, which makes it a feasible and valuable data source for extracting code-description mappings for automated comment generation [208]. Previous work had shown that Stack Overflow contains a total number of 42 million questions as of March 2017. Previous work had shown that at least 49% of the Java and Android questions in Stack Overflow have at least one code example in the accepted answer [209]. Android code snippets have a mean size of 16.4 lines of code (LOC) and a median of 9 LOC [210].

The idea is to generate comments automatically by mining Q&A sites and open-source projects. The prototype, *CloCom+*, searches for similar code segments between the two sources against the target software project. Once CloCom+ finds an identical or similar code segment, it presents the corresponding description as a comment to explain the matched code segment. One key benefit of CloCom+ is that *the description is what a developer uses to describe the code segment*, which is likely to be accurate and useful for developers to understand code (compared to descriptions generated from variable names and method names). However, our evaluation shows that CloCom+ is limited by the number of comments that it can generate since it relies on discovering a similar piece of the code segment that contains a natural language description. Lastly, our user study shows that the quality of the automatically generated code comments by CloCom+ is not good enough to be directly applied as a source code comment. The 20 participants could not reach a good agreement on the quality of the code comments. Although CloCom+ improves on SumSlice, it still requires improvement before it can be applied in the real world. We studied the automatically generated code comments and proposed several future directions for automatic comment generation.

The paper makes the following contributions:

- We proposed and built our tool, CloCom+, as an open source tool shared on our website [211], that is capable of generating code comments at a code fragment level where the code segment contains an average of 3—4 statements.

- We evaluated CloCom+ on 16 open source projects, and it successfully generated 442 groups of unique comments for 780 code locations. We performed a user study with 20 participants to evaluate the quality of the automatically generated comments by CloCom+ and compared it against the latest state of the art. Although the result shows that the majority of the participants found the automatically generated comments are complete, concise, expressive and useful, the statistical test shows little agreement between the participants. The technique is shown to be not ready for real-world usage and still require much improvement.
- Our comparison against previous work, SumSlice, shows that the natural language summary that previous work generates from the method signature greatly improves the natural language summary's completeness. However, certain sentences that are included in their generated summary caused their technique to be overall less concise compared to CloCom+.

5.2 Examples and Challenges

In this section, we present three examples to illustrate how CloCom+ generates comments automatically. We describe the challenges, summarize our solutions, and show comparisons against previous work by Sridhara et al. and McMillan et al.

5.2.1 Example One

Figure 5.1 shows a code segment from the Java project—Eclipse. CloCom+ generates the following comment to explain the code segment between line 13—15: *"**Create** a **temp file** first."*

Figure 5.2 shows the Stack Overflow post that CloCom+ leverages to generate the comment. It shows the title of the post, the code snippet, and one paragraph immediately before the code snippet in the answer.

Challenges in Comment Selection: Figure 5.2 shows two textual descriptions that can be leveraged to describe the code segment in the answer. One is the title of the post, which describes the question. The other is the paragraph immediately before the code segment, which consists of a single sentence. Among the two sentences in the title and the answer paragraph, only the sentence in the answer describes the code snippet in Figure 5.2. CloCom+ needs to select this relevant sentence from the two sentences to generate the comment automatically, which is challenging.

```
1  private File getLocalCopy(File file) throws IOException {
2      String fileName = file.getName();
3      String prefix;
4      String suffix = null;
5      int dotLoc = fileName.indexOf('.');
6      if (dotLoc = -1) {
7          prefix = fileName.substring(0, dotLoc);
8          suffix = fileName.substring(dotLoc);
9      } else {
10         prefix = fileName;
11     }
12
13     File tmpFile = File.createTempFile(prefix, suffix);
14     tmpFile.deleteOnExit();
15     FileOutputStream fos = new FileOutputStream(tmpFile);
16     FileInputStream fis = new FileInputStream(file);
17     byte[] cbuffer = new byte[1024];
18     int read = 0;
19
20     while (read = -1) {
21         read = fis.read(cbuffer);
22         if (read = -1)
23             fos.write(cbuffer, 0, read);
24     }
25     fos.flush();
26     fos.close();
27     fis.close();
28     tmpFile.setReadOnly();
29     return tmpFile;
30 }
```

Figure 5.1: Code from Java project—Eclipse (PluginsView.java).

Stack Overflow Question (Title):
*Is it possible to **create** a **File** object from InputStream*
Stack Overflow Answer:
***Create** a **temp file** first.*

```
1 File tempFile = File.createTempFile(prefix, suffix);
2 tempFile.deleteOnExit();
3 FileOutputStream out = new FileOutputStream(tempFile);
4 IOUtils.copy(in, out);
5 return tempFile;
```

Figure 5.2: Stack Overflow Post #11501418

CloCom+ uses a text similarity technique to address this comment selection challenge. Some sentences extracted from Stack Overflow do not actually describe the code segment and has to be modified. CloCom+ leverages the *text similarity* between each sentence and the code segment to identify the most relevant sentences (Section 5.3.5). In Figure 5.2, the shared words between the text and code are in bold—**create**, **file** and **temp**, which results in a text similarity score of two in the title of the post, and a score of three in the answer's sentence. CloCom+ discards the title of the post because it has a lower text similarity score.

Previous work generates comments for high-level actions within methods [90]. Their work generates comments for three types of statement groups. The first type involves sequences of statements that can be grouped into a high-level action (same verb phrase with a common headword). The second type involves conditional blocks that contain integrable sequences of statements along the branches (`if ... else if ... else ...` or `switch`). The third type involves specific common high-level code patterns that are based on loop constructs. Their work cannot generate a comment for this code segment. The code contains no sequences of statements that can be grouped together. For example, method `createTempFile()` from line 13 and `deleteOnExit()` from line 14 contain different actions, `create` and `delete`. The code also does not contain any supported conditionals or loop constructs. For example, the `if ... else` condition and `while` loop are both not supported.

Previous work generates summaries for Java methods [89], which are leading comments that occur before a method. For example, it can generate a summary for the method, `getLocalCopy`, under Figure 5.2. Their technique first selects important lines of code that are important to a method summary through their S_unit selection process. It selects the return statement on line 29 because it is the exiting point of a method; selects

line 14, 23, 25, 26, 27 and 28 because they contain method calls that do not have a return value; it selects line 2 because the method, `getName`, contains the same action as method, `getLocalCopy`; and selects line 22 because the if-statement controls a previously selected S_unit at line 23. It then performs filtering and removes line 26 and 27 because they contain the keyword, `close`. In the end, it synthesizes natural language sentences for line 14, 23, 25, 28 using their Software Word Usage Model, which then becomes the method summary. Our work focuses on synthesizing comments for high-level actions within methods instead of synthesizing comments for the entire method.

Recent previous state-of-the-art by McMillan et al. generates a summary for Java methods [207]. It breaks down the method signature using SWUM and first generates a sentence that describes the purpose of the method ("*This method gets a local copy of the file*"). It then attempts to generate a sentence that describes the usage of the return value (e.g., "*That file is used in methods that...*"), and generates a sentence that describes the caller of the method (e.g., "*Called from method that...*"). Lastly, it utilizes PageRank algorithm to determine the importance of the sentence (e.g., "*getLocalCopy() seems far more important than average because it is called by X methods*"). Their work focuses on providing information about the context of the method, whereas CloCom+ focuses on generating code comments for the statements inside a method.

Existing text retrieval techniques on code summarization [98, 99, 100, 101] can detect important terms such as `file`, `temp`, `create`, `stream` and `delete`. However, they only generate top K terms to describe the code as opposed to natural language sentences.

Challenges in Comment Refinement: The sentences from question titles and the answers are often in a question form (e.g., "*How to ...?*", "*Is it possible to ...?*") or contain unneeded information (e.g., "*You can ...*"). The direct use of these sentences will lead to low-quality comments.

To address this challenge, we deploy natural language processing (NLP) techniques to extract the core parts of a sentence. CloCom+ looks for a subtree that contains a verb phrase (VP) and a noun phrase (NP) from the parse tree of a sentence. In Figure 5.2, CloCom+ extracts "*Create a File object from InputStream*" from the title. It removed "*Is it possible to*" from the beginning of the sentence because the leading verb is "create." It also removes personal pronouns from the leading part of the sentence if it exists (Section 5.3.2).

5.2.2 Example Two

Figure 5.3 shows a code segment from the Java project—JabRef. CloCom+ generates the following comment for the top code segment between line 3—6: *"try **parsing** as a*

```
1 public static Optional<Month> parse(String value) {
2     ...
3     try {
4         int number = Integer.parseInt(value);
5         return Month.getMonthByNumber(number);
6     } catch (NumberFormatException e) {
7         return Optional.empty();
8     }
9 }
```

```
1 private int parseMonth(String month) throws
     IllegalArgumentException {
2     // try parsing as a number
3     try {
4         int result = Integer.parseInt(month);
5         return result;
6     } catch (NumberFormatException e) {}
7
8     // try parsing as a word
9     month = month.toLowerCase();
10    if (month.equals("january") || month.equals("jan")) {
11    ...
12 }
```

Figure 5.3: Top code from Java project—Jabref (Month.java) matched against the bottom code from open source project—mdrill from GitHub (CustomPeriodicTest.java).

number." The code is matched against the bottom code segment between line 3—6. CloCom+ detects that the two code segments match.

Challenges in Code Matching: Finding similar code segments between the (1) input projects and (2) Stack Overflow and open-source projects requires code clone detection techniques (Section 5.3.3) that are scalable [36]. The detection of code clones is challenging because the code segments from Stack Overflow are often incomplete and uncompilable. To alleviate this problem, CloCom+ appends the code with dummy wrapper code (e.g., placing statements inside a main function or class) to make the code segments compilable. If the compilation is successful, it tokenizes the abstract syntax tree (AST) for code clone detection.

The two code segments in Figure 5.3 are slightly different regarding the variable names.

The renamed variables had been underlined (e.g., `number` vs. `value`). The code clone detection tool also has to capture the fact that the variable `number` is an int and `value` is a string during the matching process. On top of that, the two code segments in Figure 5.3 have a different structure, where line 5 in the top code segment does not match line 5 in the bottom code segment.

However, such small differences do not affect the semantic similarity of the two pieces of code. Therefore, the code clone detection tool ignores such differences and reports it as a code clone.

Previous work by Pollock et al. [90] cannot generate a comment for this code segment. The code contains variable declarations along with a try block and a catch block. Since there is only a single statement within each block, try block and catch block, the statements cannot be grouped together. The code also does not have any conditional or loop constructs.

Recent previous work by McMillan et al. [207] can generate a natural language summary for this Java method. However, it cannot generate a code comment for the statements within the method.

5.2.3 Example Three

Figure 5.4 shows a code segment from the Java project—Vuze. CloCom+ generates the following comment for the lines between line 3—6: *"Fall back to **system** properties."* The bottom code segment in Figure 5.4 shows the post that CloCom+ leverages to generate the comment.

Benefits of CloCom+: CloCom+ generates a comment to provide valuable information that is not explicitly in the code, e.g., the code is falling back to a value. Also, CloCom+ can naturally group the three statements into a semantic unit for comment generation because developers have already grouped the statements in the Stack Overflow post. Such grouping is general because it does not rely on the quality of the method names or the structure of the methods, which is different from previous work [90].

Previous work cannot generate a comment for this code segment [90]. The code contains no statements that can be grouped or conditional blocks. Lastly, the if-statements do not satisfy any of the supported loop patterns.

```
1 private static UnchokerFactory getSingleton(String
     explicit_implementation) {
2     String  impl = explicit_implementation;
3     if ( impl == null ){
4         impl = System.getProperty( DEFAULT_MANAGER );
5     }
6     if ( impl == null ){
7         impl    = DEFAULT_MANAGER;
8     }
9     ...
```

```
1 public String resolvePlaceholder(String placeholderName) {
2     try {
3         String propVal = this.servletContext.
              getInitParameter(placeholderName);
4         if (propVal == null) {
5             // Fall back to system properties.
6             propVal = System.getProperty(placeholderName);
7             if (propVal == null) {
8                 // Fall back to searching the system
                      environment.
9                 propVal = System.getenv(placeholderName);
10            }
11        }
12        ...
```

Figure 5.4: Top code from Java project—Vuze (UnchokerFactory.java) matched against bottom code from GitHub project spring-framework (ServletContextPropertyUtils.java).

5.3 Design and Implementation

Figure 5.5 shows an overview of CloCom+. CloCom+ takes two inputs: (1) a Stack Overflow database and/or GitHub projects; and (2) source code of the target project that user wants to generate comments on. The output is a list of generated comments.

CloCom+ consists of two major components. The first component generates databases of code-description mappings (Section 5.3.1) and leverages natural language processing (NLP) techniques to refine the descriptions (Section 5.3.2) that are from Stack Overflow. The second component generates comments for the target software. It applies a code clone detection techniques to identify matched code between the database and the target software (Section 5.3.3). It then prunes the bad matches (Section 5.3.4). Lastly, it selects

the best comment for the matched code (Section 5.3.5).

5.3.1 Code-Description Mapping Extraction from Stack Overflow

To build databases of code-description mappings from Question and Answer sites, we choose a programming website called Stack Overflow [31] as the data source. We utilize Stack Overflow's public data dump [212] to build the database. Previous work by Ponzanelli et al. presented a dataset that models each post in the Stack Overflow data dump [213]. However, we do not utilize their dataset because we can access all the required information directly from the public data dump. Stack Overflow contains questions from diverse software domains such as Java, Android, and C++. Each domain is associated with its respective tag. We built a code-description mapping database for Java projects by locating questions that are tagged with either "`java`" or "`android`." The two tags returned a total of 257,504 code-description mappings in our evaluation.

Stack Overflow contains invalid and low-quality questions and answers. Previous work by Ponzanelli et al. [214] proposed a technique to identify low-quality posts by using both the content of a post and community-related aspects. To ensure the quality of extracted code-description mappings, CloCom+ selects questions and answers based on the score it received from the voting system in Stack Overflow. For each post, CloCom+ extracts code-description mappings from all *questions* and *answers* that had received a positive (one or more) score. Figure 5.6 shows the frequency distribution of the score for all 800,744 Android tagged questions in 2016, which has a mean of 1.6 and a median of 0. We observe that 644,575 (80.5%) of the Android questions have a score between zero and two, and only 48,289 (7.5%) of the Java questions have a negative score.

The paragraph before a code segment is not the only description for the code segment

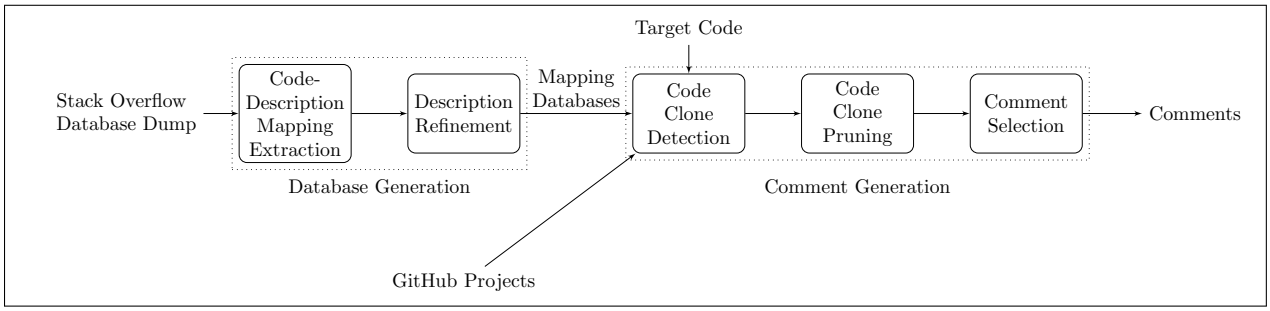


Figure 5.5: An overview of CloCom+.

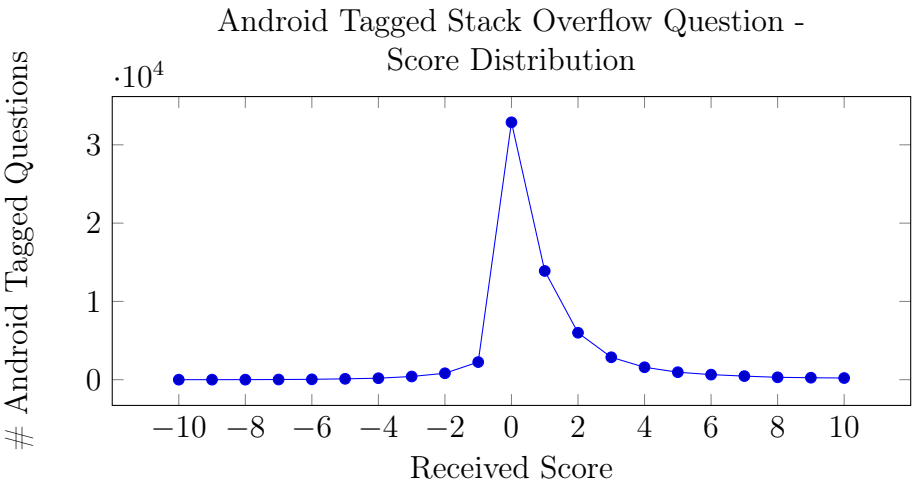


Figure 5.6: A Score Distribution of the 800,744 Android-tagged Stack Overflow questions. We do not show the distribution where a question has a score of more than ten (1602 in total) or less than negative ten (two in total).

(Figure 5.2). Since it is common for people to summarize the problem using the title of a post, we also extract the title as a comment candidate. We attempted to extract description sentences from other parts of a post. The initial theory was that analyzing more sentences will help generate more comments. Therefore, instead of only extracting from the paragraph immediately before the code segment, we tried to include description sentences from 1) two paragraphs before the code segment, 2) one paragraph after the code segment (only if there are no code segments following the paragraph), and 3) one paragraph before and one paragraph after the code segment (only if there are no code segments following the paragraph). The inclusion of additional paragraphs into the analysis can improve the yield, but the majority of the description sentences from the additional paragraphs are not describing the code segment.

Based on the above experiment, we generate the mappings by including natural language sentences from (1) one paragraph before the code segment and (2) the title of the post, which provided 257,504 code-description mappings for Java in the evaluation.

5.3.2 Description Refinement

The description sentences of the code segments are often in a question form (e.g., "*How can I ...?*") or contain unnecessary information (e.g., "*You can try ...*"). To improve the quality of descriptions, CloCom+ leverages NLP techniques to perform refinements. It

Table 5.1: List of terms for sentence filtering.

no, not, cannot, but, or, bad, fail, error, errors, exception, fix, bug, missing, help, wrong, problem, efficient, expensive, slow, incorrect, instead, likely, maybe, seems, perhaps, probably, think, where, why, crash, defect, patch, how, just, really

performs filtering of invalid descriptions and extraction of core parts of the descriptions. CloCom+ refines the descriptions using techniques in the following order.

Description Filtering: As discussed in Section 5.2.1, sentences that ask and answer how to troubleshoot code often do not describe the code segment. For example, "*Why this code does not work?*" and "*Android: problem retrieving bitmap from database.*" CloCom+ filters out such sentences based on the manually collected terms in Table 5.1. This list was collected through manual inspection of the extracted sentences from Section 5.3.1. We attempt to generalize this list through expanding the terms with synonyms (e.g., "maybe" to "perhaps"). We use the same list for filtering both the Java and Android databases. CloCom+ removes a sentence from being a potential comment candidate if any of the terms in the list appears in the sentence.

We tried to apply Stanford's sentiment analysis tool to detect negative sentences, but most of the sentences were being classified incorrectly and hence the idea had been discarded. In the future, it may be interesting to determine if a sentence contains negative or troubleshooting information using techniques such as semantic role labeling to retrieve high-level information associated with the verb (action) of a sentence.

Main Subtree Extraction: Sentences that are in a question form or contain personal pronouns (e.g., "you") are not suitable as comments. Therefore, we adapt NLP techniques with two objectives: 1) to convert the sentences in a question form to a statement form, and 2) to extract the core parts of the sentences. We achieve them by identifying and extracting the main subtree of a sentence.

There are three steps to extract the main subtree of a sentence:

1. Generate a parse tree from the input sentence.
2. Obtain all the subtrees that match the specified patterns.
3. Merge all the matched subtrees together to form a refined sentence.

Step one generates a parse tree using the latest version of Stanford CoreNLP[1] (v3.6.0). CloCom+ first uses CoreNLP's part-of-speech (POS) tagger [34] to label the part of speech of the words in each sentence. It then uses the statistical parser [35] to generate the parse tree. Figure 5.7 shows the parse tree for a sentence in a Stack Overflow post. The leaf nodes represent the words of the sentence, and the parent of each leaf node represents the POS tag of the word.

Since sentences in the Stack Overflow post are from the software domain, it is very common for them to contain code artifacts (e.g., `ClassName.methodName()`). Code artifacts cause CoreNLP's POS tagger to fail at tokenizing sentences correctly. To mitigate this issue, we identify such code artifacts using simple regular expressions and correct them as a noun before the tokenization process.

Step two extracts the main subtrees from the parse tree. The idea is to obtain subtree(s) that contain at least one verb phrase (VP) and one noun phrase (NP). It ensures each extracted phrase contains a verb and a subject/object. We define two patterns, Equation (5.1) and Equation (5.2), in Stanford's Tregex[2] format to extract the main subtree(s) of a parse tree. Table 5.2 and Table 5.3 explain these two patterns respectively. We formulated the two patterns based on manual inspection of hundreds of natural language sentences in Stack Overflow to ensure the generality of the two patterns. Since the two patterns are only defined on a verb/noun phrase level, they have a high compatibility towards a wide range of sentences.

We require the definition of two different Tregex patterns for two reasons. First, each pattern contains a unique set of constraints on top of requiring having at least one VP and NP. For example, the VP-NP (a verb phrase followed by a noun phrase) pattern in Table 5.2 contains the following two constraints. 1) The NP inside the VP must not only contain a personal pronoun, and 2) the VP must not only contain a modal verb. Second, each pattern defines the unique grammatical structure layout of the verb/noun phrase (e.g., ancestor and sister relationship in the parse tree), which cannot be defined using traditional regular expression matching.

$$\textbf{VP-NP} : VP << (NP < /NN.?/) < /VB.?/ \tag{5.1}$$

$$\textbf{NP-VP} : NP\,! < PRP\,[<< VP \,|\, \$\,VP] \tag{5.2}$$

The NP-VP pattern excludes noun phrases that begin with a personal pronoun (PRP). Personal pronouns typically contribute no value in a code comment, so it is safe to remove

[1] http://stanfordnlp.github.io/CoreNLP/
[2] http://nlp.stanford.edu/software/tregex.shtml

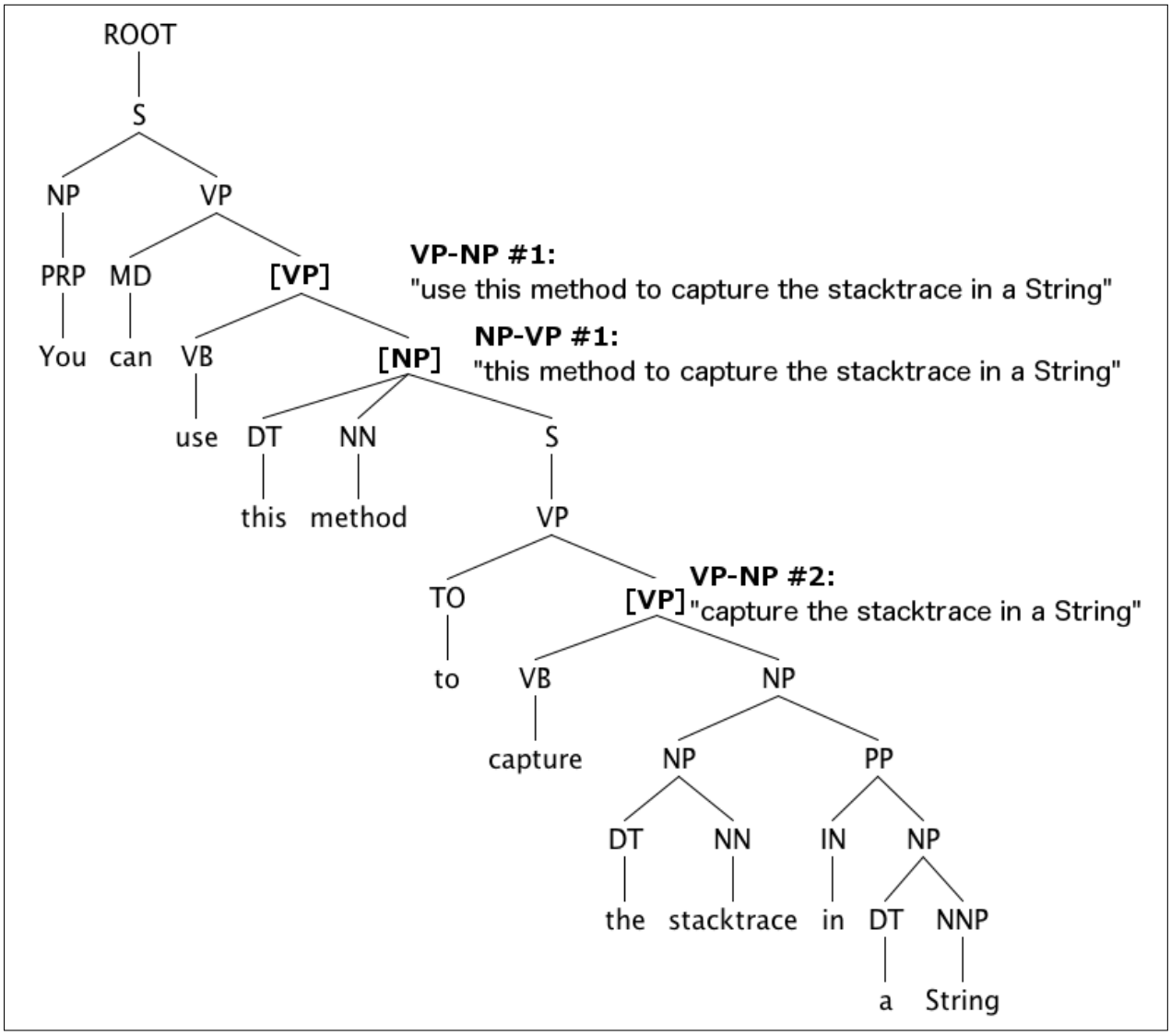


Figure 5.7: Parse tree for the sentence "*You can use this method to capture the stacktrace in a String.*" The matched Tregex patterns are labeled in bold.

Table 5.2: Explanation for Equation (5.1), VP-NP.

Regular Expression	**Explanation**	**Rationale**
VP << (NP < /NN.?/)	Verb phrase (VP) that is an ancestor of a noun phrase (NP).	Ensures the sentence starts with a VP that includes an NP.
NP < /NN.?/	Noun phrase (NP) that is the parent of the basic category of a noun (NN).	Ensures the NP has at least one noun that is not a personal pronoun (PRP), but the NP will be allowed to contain personal pronouns.
VP < /VB.?/	Verb phrase (VP) that is the parent of the basic category of a verb (VB).	Ensures the VP has at least one verb that is not a modal verb (e.g., can, must, should, will).

them. Penn Treebank tag guideline [215] defines PRPs to include personal pronouns proper (i.e., "I", "me", "you", "he", "him", etc.), reflexive pronouns ending in *-self* or *-selves*, and nominal possessive pronouns (e.g., "mine", "yours", "his", "ours" and "theirs").

We show how to remove "You can" from the sentence in Figure 5.7 using the two patterns. CloCom+ finds three subtrees that are matched by the patterns, and then merges them (step three), which produces a sentence without "You can." We label the matched VP-NP and NP-VP patterns on the right-hand side of the figure. The conditions that each matched subtree have satisfied are as follows:
VP-NP #1: "use this method to capture the stacktrace in a String"

1. The VP (highlighted as [VP]) is an ancestor of the NP, "this method to capture the stacktrace in a String."
2. The NP is the parent of a noun, "method."
3. The VP is the parent of a verb, "use."

NP-VP #1: "this method to capture the stacktrace in a String"

1. The NP (highlighted as [NP]) is not the parent of a personal pronoun.
2. The NP is an ancestor of the VP, "to capture the stacktrace in a String."

Table 5.3: Explanation for Equation (5.2), NP-VP.

Regular Expression	**Explanation**	**Rationale**
NP !<PRP	Noun phrase (NP) that is not the parent of a personal pronoun (PRP).	NP that is a personal pronoun offers no value to the comment, thus excluded.
NP [<< VP | $ VP]	Noun phrase (NP) that is either the ancestor or sister of a verb phrase (VP).	Ensures the sentence starts with an NP followed by a VP. VP that appears after an NP often appear on the same level in a parse tree.

VP-NP #2: "capture the stacktrace in a String"

1. The VP (highlighted as [VP]) is an ancestor of the NP, "the stacktrace in a String."
2. The NP is the parent of a noun, "stacktrace."
3. The VP is the parent of a verb, "capture."

For the other sentence in the motivating example, "*Is it possible in Java's MessageFormat to receive a stack trace?*", CloCom+ extracts the main subtree as "Receive a stack trace" because it matches with Equation (5.1).

Step three performs merging on the extracted subtrees in the case where there is more than one subtree and outputs a single sentence. For example, the parse tree in Figure 5.7 contains three matched subtrees (VP-NP #1, NP-VP #1, and VP-NP #2). To generate a single sentence from the multiple matched subtrees, CloCom+ calls a method from Stanford NLP's tree class, "join node," on all the subtrees which performs the following: *Given two subtrees, locate node j such that j dominates both subtrees, and return a tree with node j as the root of the tree.* In other words, we locate an English phrase that contains both of the matched phrases (or matched subtrees). In this example, since the first subtree dominates the second and third subtrees, the "join node" operation returns the first subtree as the generated comment.

The persudo code for merging the sub-trees is shown in Listing 5.1. To generate a single sentence from the matched subtrees in step two (NP-VP and VP-NP phrases), CloCom+ merges all the subtrees by calling an existing method "joinNode." The method locates a node in the given parse tree that dominates the given two subtrees, and it returns the dominate node of the two subtrees as the merged tree.

```
1  // Obtain the parse tree of the input sentence
2  parseTree = getParseTree(inputSentence);
3  // Locate VP-NP and NP-VP patterns in the tree
4  vpnp = getTrees(parseTree, "VP<<(NP</NN.?/)</VB.?/");
5  npvp = getTrees(parseTree, "NP!<PRP[<<VP|\$VP]");
6  // merge the found sub-trees
7  mergedTree = null;
8  while (vpnp.found()) {
9    mergedTree = parseTree.joinNode(mergedTree, vpnp.getTree());
10 }
11 while (npvp.found()) {
12   mergedTree = parseTree.joinNode(mergedTree, npvp.getTree());
13 }
14 mergedSentence = mergedTree.toSentence();
15 return mergedSentence;
```

Listing 5.1: Natural language tree merging algorithm.

5.3.3 Code Clone Detection

We utilized an existing token-based clone detection tool that was implemented in our previous work, CloCom [7], to detect matched code segments between Stack Overflow and GitHub projects, against the target project from the user. However, other existing code clone detection tools that perform subtree analysis on abstract syntax trees (ASTs), or control and data flow analysis on program dependence graphs (PDGs) can also be utilized for clone detection. Previous work summarized the differences between the existing tools [118]. CloCom's code clone detection algorithm is based on DuDe's algorithm [119], which is a code clone detection tool based on the scatter plot algorithm. CloCom compiles the input source code into an AST and transforms the AST into tokens to detect code clones.

Listing 5.2 shows the pseudo-code of the code clone detection algorithm. The algorithm first obtains a list of methods from each file, and all the hashed statements within each method. The scatter plot is then populated based on the hash numbers. The remainder of the algorithm is the same as DuDe's algorithm where it detects non-gapped clones and the gap locations, and builds the longest chain of code clone that does not exceed the maximum gap size. Lastly, it adds the clone chains that satisfies the minimum length as a code clone.

CloCom provides a simple set of low-level AST tokenization rules for code clone detection. These rules provide a tradeoff between accuracy and yield of the tool. For example,

```
1  for method1 in file1:
2    for method2 in file2
3      statements1 = method1.getStatements();
4      statements2 = method2.getStatements();
5      for statement1 in statements1:
6        for statement2 in statements2:
7          if statement1.hashNumber() == statement2.hashNumber():
8            populateScatterPlot();
9          end
10     end
11   end
12 end
13 chainList = detectNonGappedCloneChains(scatterPlot);
14 gapMap = buildGapMap(chainList);
15 listChains = buildLongestChain(chainList,gapMap,maxGapSize);
16 for chain in listChains:
17   if chain.length() > minLength:
18     recordAsClone();
19   end
20 end
```

Listing 5.2: Clode clone detection tool's algorithm.

allowing variable name differences between code clones significantly increases the yield while having a negligible impact on the accuracy.

CloCom requires the code segment to be compilable by the Eclipse Java compiler to extract the AST of the code. Most clone detection tools, except for token-based clone detection tools that do not rely on the AST for tokenization (e.g., SIM [112]), require compilable code. Stack Overflow contains code segments that are not wrapped inside a class or method, which makes them uncompilable. Therefore, we implemented a simple heuristic similar to previous work [216], which wraps the code segment with a method or class to ensure it is compilable. CloCom+ attempts to repair a code segment if it fails to compile. It checks for `import` statement, `package` statement, `class` signature, and `method` signature in the code segment. If the code is missing a method signature, it appends "`public class A {public static void main(String[] args) {`" and "`}`" to the front and back of the code segment respectively. Similarly, if the code is missing a class signature, it appends "`public class A {`" and "`}`" to the code segment. It can repair the majority of the code segments into a compilable state for compilation.

In the matching algorithm, CloCom supports clones that contain gaps [217]. Figure 5.8

```
1   double d2;
2   try {
3   d2 = Double.parseDouble(conditionValue);
4 } catch (NumberFormatException fne) {
```

```
1 double result;
2 try {
3   result = Double.parseDouble(testString);
4   System.out.println("Success!")
5   }
6 catch (NumberFormatException nfe) {
```

Figure 5.8: Top code segment from Java project Freemind matched against the bottom code segment from Stack Overflow (post #35109636) depict the highlighted missing statement between the two code segments. CloCom+ generated "Check if a string is a floating point number." to describe the source code comment.

shows a code segment from project FreeMind that is matched against Stack Overflow post #35109636, where four statements had been matched between the two code segments. The two code segments contain one gap. Statement 1-3 from the Stack Overflow code segment is connected to statement six with a gap in line four. The gap in the Stack Overflow code segment contains an extra statement that invokes `println()`. We configured the matching algorithm to allow a maximum gap size of two for each code clone pair, which allows a maximum of one modified or deleted (or equivalently inserted) statement at each gap location.

We configured the gap size threshold to be as high as possible without sacrificing the quality of the obtained code clones. A gap size that is too high can render the comment from Stack Overflow invalid due to the additional statements. We limit the number of altered statements to one statement at each gap point.

We configure CloCom to report clones that contain a minimum of four matched statements. The threshold that controls the code clone's size is `minCloneSize`. A smaller clone size requirement increases the number of detected code clones, but it also increases the number of false positives the clones will contain less content. We studied the impact of having different values of `minCloneSize` on the yield. Figure 5.9 shows an experiment when the size of all the returned code segments when `minCloneSize` is configured to three statements. Since a large number of the code segments (2,220) contain a size of three statements, we selected a clone size of three as the threshold for the evaluation in this

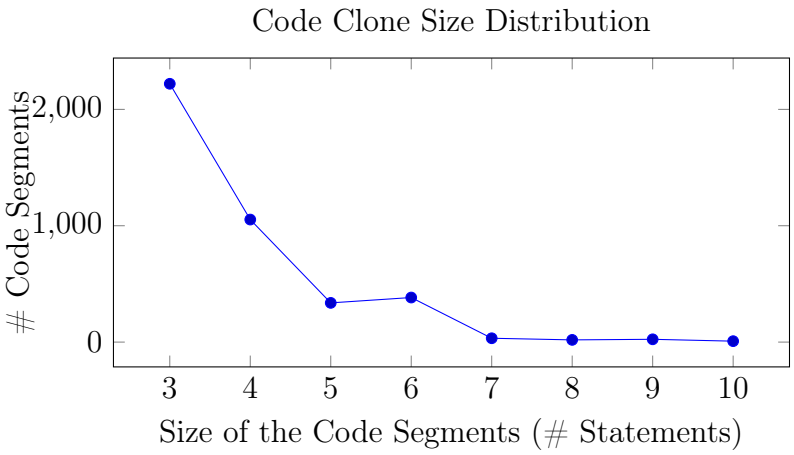


Figure 5.9: Code clones' size distribution in Stack Overflow.

work.

5.3.4 Code Clone Pruning

The output of the code clone detection tool consists of pairs of code segments that have a similar syntactical structure. It is important to ensure a high level of semantic matching to generate accurate and useful comments.

Line Percentage Matching (Stack Overflow only): To ensure the extracted description sentence applies to the matched code segment in the target software, we have to ensure a high proportion of the Stack Overflow code segment is matched against the code in the target project. Therefore, CloCom+ calculates the percentage matching score as a filtering metric.

Specifically, for each Stack Overflow code segment, we count the number of lines in the extracted code segment (`cTotal`) and the number of lines that had been matched by the code clone detection tool (`cMatched`). CloCom+ calculates the percentage matching score (`PMS`) using Equation 5.3 with a 60% threshold. At least 60% of the effective lines have to match. We select this threshold because it allows toleration between 1-2 lines of content for small code segments. For example, if a Stack Overflow code segment contains five lines of code, the threshold would require the matching of at least three lines of code. We configure this threshold based on our observation that matching less than 60% of the effective lines of code often renders the source code comment invalid. The reason is that a

```
1 int i;
2 int j;
3 int k;
```

Figure 5.10: Example of a repetitive clone.

lower threshold means more code is not matched during the code clone detection process.

$$\text{PMS} = \frac{\text{cMatched}}{\text{cTotal}} \tag{5.3}$$

We varied the threshold with different values (40%, 60%, and 80%) on a Java project, GanttProject, which generated 12, 8 and 5 unique clone groups respectively. We manually classified the generated comments within each clone group and determined that 60% provides the best tradeoff between yield and quality. At 40%, 8 out of the 12 comments do not describe the code; at 60%, 4 of the 8 comments do not describe the code; at 80%, 1 of the 5 comments do not describe the code.

Repetitive Clone Pruning:

CloCom+ removes code clones that contain repeated statements, which commonly occur in variable declaration code. Such code is not useful as a code clone and is often heavily reported by the code clone detection tool. The pruning algorithm checks the hash value of the statement to determine if two statements are the same. It removes code clones that contain repeated statements. Statements that are the same in structure but differs in the variable names are the only code that will be removed from this heuristic. In Figure 5.10, since our code clone detection tool tolerates variations on the variable name, the three statements are considered to be the same. Therefore, CloCom+ removes the code clone.

Other Filters: To ensure high semantic matching, CloCom+ requires the matched code segment to contain at least one statement that performs a method invocation. Furthermore, CloCom+ prunes code matches that contain over ten source code statements because they often contain multiple semantic units, and Stack Overflow is unlikely to contain detailed enough descriptions.

5.3.5 Comment Selection

For each remaining match, there can be one or more description sentences from the title and body of the post available as a comment candidate. If the code from the target

project matches with multiple Stack Overflow code segments, CloCom+ considers all of the description sentences of each Stack Overflow code segment as a candidate and selects the best comment. Do note that if an automatically generated comment already exists in the target project, CloCom+ automatically removes it from the candidate pool.

Text Similarity: To select the best description sentences from the remaining sentences, CloCom+ measures the *text similarity* between each candidate description sentence against the code segment in the target software. Algorithm 2 describes the steps to perform text similarity for each clone group. A clone group refers to a group of code segments that contain a similar structure identified by the code clone detection tool. The goal is to obtain a list of comment candidate that satisfies the text similarity requirement (having at least one common term between the sentence and code segment). Line 2—3 first obtains the list of master clones (clones from the target project) and the list of database clones (clones from Stack Overflow and GitHub). Line 4 iterates through each database clone. Line 5 extracts a list of terms from the code segment of the database clone. Specifically, it retrieves all the identifiers (e.g., variable names and method names) within the method that surrounds the code segment. Line 6 obtains the list of comments in the current database clone and proceeds to obtain a list of stemmed words of the sentence in line 8. Text similarity relies on obtaining a list of common terms between the code segment and the words in the code comment (line 9). The common terms between the two must exist on each master clone (line 11) to ensure the comment candidate is not specific to the database clone. If the common terms exist in all of the master clones (line 19), then the comment is added to the candidate list (line 20).

5.4 Evaluation Method

We conducted a user study[3] with externally recruited participants (engineering students and industrial programmers) to evaluate the quality of the comments generated by CloCom+ and a recent previous work, SumSlice [207].

RQ1: Are the automatically generated comments by CloCom+ *complete*, *concise*, *expressive*, and *useful* in describing the code?

The research question evaluates the baseline quality of the automatically generated comments similar to a recent previous work on automatic unit test case documentation [218].

[3]This research has received ethics clearance approval (#18754) from University of Waterloo's Office of Research Ethics.

Algorithm 2: Algorithm for text similarity.

```
Input  : cloneGroup
Output : commentCandidateList
commentCandidateList ← null
masterCloneList ← masterClonesInCloneGroup(cloneGroup)
databaseCloneList ← databaseClonesInCloneGroup(cloneGroup)
for databaseCloneList ∈ databaseCloneList do
    DC_terms ← getSimpleTerms(databaseCloneList)
    listComments ← getListComments(databaseCloneList)
    for sentence ∈ listComments do
        wordList ← getListStemmedWords(sentence)
        mustExistTerms ← getCommonTerms(DC_terms, wordList)
        satisfiedRequirement ← true
        for masterClone ∈ masterCloneList do
            MC_terms ← getSimpleTerms(masterClone)
            status ← allTermsExists(mustExistTerms, MC_terms)
            if status = false then
                satisfiedRequirement ← true
                break
            end
        end
        if satisfiedRequirement = true then
            commentCandidateList ← add(sentence)
        end
    end
end
```

It rates a comment based on its completeness (i.e., does not miss important information), conciseness (i.e., not containing redundant or useless information), expressiveness (i.e., readability), and usefulness (i.e., helps developers understand the code). We added the usefulness criteria which the previous work did not evaluate. The usefulness criterion refers to the value of having the comment present to understand the source code. We evaluate the usefulness criterion because a comment can be complete, concise, and expressive, but does not help developers understand the code.

> Majority of the participants rated strong agree or weakly agree on the completeness, conciseness, expressiveness, and usefulness of the generated comments. However, the statistical tests failed to show agreement between the participants. The automatically generated comments are not ready for real-world usage due to the lack of quality consistency.

RQ2: Are the automatically generated comments by SumSlice *complete*, *concise*, *expressive*, and *useful* in describing the code?

The research question allows us to contrast our work against a recent previous work, SumSlice [207]. SumSlice generates code summaries for Java methods, which is the closest piece of work against CloCom+. The evaluation approach follows RQ1's criteria and steps.

> SumSlice is capable of generating method summaries that are more complete at describing the code segment. Specifically, the natural language summary that is generated based on the method signature provides a clear explanation of the code segment. However, the generated summary is not as concise as CloCom+ since the summary includes additional sentences that are not as useful to the participants (i.e., sentences that describes the importance of a method using PageRank).

RQ3: What are the participants' opinion on the advantages and shortcomings of CloCom+ and SumSlice?

The research question is conducted as a post-study to understand the pros and cons between CloCom+ and SumSlice. Unlike RQ1 and RQ2 which are formulated as a quantitative study based on a 4-point Likert scale, we study the participants' qualitative response.

For CloCom+, the majority of the participants expressed concerns about its consistency at generating quality code comments. However, they seem to welcome the ability to link a Stack Overflow post against the target code segment. For SumSlice, the majority of the participants agree that the method summaries are clear and complete at describing the code. Specifically, the ability to generate a natural language sentence from the method signature. Several had expressed concerns of the summary being a pure rephrase of the code and that the summary is not very concise.

RQ4: What are the reasons that the automatically generated code comment is not applicable to the code segment?

The research question attempts to understand the reason that causes a weak rating on the completeness, conciseness, expressiveness, and usefulness criteria.

In term of completeness, the weak score is due to the comment containing missing or useless information. Specifically, CloCom+ fails to generate comments that contain contextual information. In conciseness, the weak score is due to the sentence being informally written. In expressiveness, it is due to the sentence being hard to understand or contain wrong information. In usefulness, it is due to the failure to filter out trivial comments.

RQ5: What are the properties of the developer-written and participant-written comments?

The research question attempts to understand the properties of human-written comments (from developers and participants in the study). The purpose is to guide designs of future automatic comment generation tools.

The result shows that amongst the manually inspected 110 JavaDoc and 17 single line developer-written comments, they all contain a maximum of two sentences. The code comments' often simply provide information about the action or purpose of the source code, and that they rarely directly reuse the identifier name. The manually inspected 135 participant-written comments have three properties. They are often a natural rephrase of the source code (84.4%), only contains a single sentence (97%), and the sentence often starts with a verb (84.4%).

5.4.1 Experimental Settings

For CloCom+, We apply it to analyze Stack Overflow questions that contain the `Java` or `Android` tag to create a code-description mapping database for a total of 257,504 code-description mappings. After applying the techniques, we had generated a total of 442 unique comments (clone groups). Table 5.5 shows the total number of lines of code calculated using CLOC [219]. For each of the evaluated open source Java projects. It also shows the number of uniquely generated comments (clone groups) per project. A clone group represents a group of code segments that are structurally similar to each other, which can all be described with the same code comment (see Section 5.3.5 for further details). We run the 16 Java projects against the `Java` database. Sridhara et al. evaluated a total of 15 Java projects in their work. In our evaluation, the 16 Java projects include the same 15 Java projects that were evaluated by previous work [90], and one additional Java project—Eclipse.

For the comparison against SumSlice [220], we downloaded the evaluation package of SumSlice from their website, which contains the generated code summary for all the methods for the Java project, NanoXML (286 summaries in total). Do note that SumSlice only generates comments for Java methods as opposed to CloCom+ which generates comments for code segments. We attempted to run CloCom+ on NanoXML, but we were not able to generate any code comments due to the limited yield of our tool. Therefore, we do not attempt to perform a direct comparison of CloCom+ against SumSlice over the same piece of source code. Instead, the comparison will be indirect since we cannot generate comments over the same code. However, we ran CloCom+ on all the Java projects that SumSlice [207] had evaluated, which includes NanoXML, Siena, JTopas, Jajuk, JEdit and JHotdraw, and show the result of these projects in Table 5.4.

5.4.2 Human Participants

The study includes 20 human participants to rate the comments generated by CloCom+. We recruited student participants from University of Waterloo's Computer Science, Software Engineering and Electrical and Computer Engineering department using the school's internal mailing list. The evaluator group includes nine graduate students, nine undergraduate students and three industrial programmers. All participants are specified to require a minimum of one year of industry programming experience. The participants have an average of four years of industrial programming experience (between 1—10 years). We remunerate each participant with a prize of 15 Canadian dollars for the completion of the study.

Table 5.4: Results of the six SumSlice projects by CloCom+.

Java Project	Lines of Code	# Generated Comments (Clone Groups)	# Commented Locations	Exec. Time (min)
JHotDraw	32,122	7	8	21
Jajuk	68,767	11	19	42
NanoXML	5,051	0	0	7
Siena	52,751	8	15	35
JTopas	9,037	0	0	10
JEdit	123,399	15	15	61
Total	291,127	41	57	176

Table 5.5: List of the 16 evaluated projects by CloCom+. CloCom+ generated 442 comments for 780 code locations in the target project. The numbers in bracket represent the result of CloCom.

Java Project	Lines of Code	# Generated Comments (Clone Groups)	# Commented Locations	Exec. Time (min)
Eclipse SDK	3,488,835	235 (84)	447 (96)	1,100
FreeCol	133,637	3 (9)	4 (15)	37
FreeMind	68,677	6 (6)	6 (9)	23
GanttProject	62,409	8 (2)	8 (4)	31
Hibernate	581,642	28 (9)	50 (12)	214
HSQLDB	168,239	43 (27)	60 (35)	80
JabRef	93,685	7 (18)	9 (20)	38
Jajuk	68,767	11 (5)	19 (8)	42
JavaHMO	25,631	8 (3)	22 (5)	18
JBidWatcher	28,357	15 (11)	17 (16)	23
JFtp	22,394	2 (2)	10 (2)	17
JHotDraw	32,122	7 (6)	8 (7)	21
MegaMek	295,412	19 (8)	31 (13)	138
Planeta	13,574	1 (1)	1 (1)	11
Sweet Home 3D	97,210	12 (9)	14 (15)	51
Vuze	613,557	37 (15)	74 (19)	257
Total	5,794,148	422 (215)	780 (277)	2101

5.4.3 Questionnaire Generation

We provide each participant with a digital questionnaire generated using Google Forms. Each questionnaire contains 15 questions. Since we require 15 opinions for each question, we recruited 15 participants for each questionnaire. In order to increase the sample size of the population, we created two questionnaires and performed the study with 20 participants. The participants are separated into two groups each with 15 participants. Group one answers the first questionnaire and group two answers the second questionnaire. Each group evaluates the same set of questions. The two questionnaires are shared on our website. Each questionnaire contains 15 generated comments (12 questions for evaluating CloCom+ and 3 questions for evaluating SumSlice [207]. We document the questionnaire generation process below. Each participant within the group receive the same 15 questions (one comment per question) to ensure we can obtain multiple opinions (10) per question. The multiple opinions allow us to measure the level of agreement between the participants with statistical tests. We randomly sample 24 comments from the 422 unique comments from CloCom+, and sample 6 comments from the 286 unique comments from SumSlice for the evaluation. Each group evaluates 12 comments for CloCom+ and 3 comments for SumSlice.

5.4.4 Study Procedure

To ensure the format of the user study is clear and easy to understand, we asked two undergrad students to perform a pilot study. At the end of the study, we collected the feedback from the students about the study. The feedback reported that all the questions and steps were easy to complete and comprehensible.

In the main study, we show the code segment and ask the participants to write a comment for the code. Asking participants to write down a comment can help them understand the code segment before rating the automatically generated comment. Since a code segment can be hard to understand without context, we show the surrounding code of the code segment to help participants understand the code segment. Participants may give up on writing a comment if they find it difficult. To avoid overwhelming the participants with an excessive amount of information, we show an average of 20 surrounding code lines per code segment. Second, we ask participants to rate the comments with a four-point Likert scale [221] on the completeness, conciseness, expressiveness, and usefulness criteria similar to previous work [222]. Following Moreno et al.'s recent work [223], we asked developers to rate the answers using a 4-point Likert scale instead of a 5-point Likert scale to avoid neutral answers. We use the following four-point Likert scale: (1) Strongly Agree,

(2) Weakly Agree, (3) Weakly Disagree, and (4) Strongly Disagree. Since CloCom+ is capable of linking a code segment to the related Stack Overflow post, we show the link to the related Stack Overflow post and ask the user to rate its usefulness whenever it is available.

Previous work by Papineni et al. [224] proposed the usage of BLEU score to evaluate the predictions that are made by automatic machine translation (MT) systems. Existing work on commit message generation [225] and automatic comment generation [96] also utilize the metric to measure the quality of the generated text. The BLEU score measures the number of matching n-grams between the candidate and reference translation. The BLEU score relies on the existence of a reference code comment as a baseline for the measurement. Since CloCom+ generates code comment for code segments within a Java method instead of generating code comment for Java methods, a reference code comment may not be always available for the calculation of the BLEU score. Therefore, we decided to use an evaluation that involves human participants [222] with a four-point Likert scale to rate the generated comments. An advantage of human participants is that they are able to help rate the usefulness of the comments, which cannot be measured by the BLEU score.

5.4.5 Post Study Questions

After the study, we reveal to the user that the first 12 questions are generated using CloCom+ (tool #1), and the last three questions are generated using SumSlice (tool #2). We asked two post study questions after the participant had completed the questionnaire to understand the utility of the technique.

PQ1: Which tool do you prefer more?

The participants may choose an answer either "Tool #1", "Tool #2", "I don't like any of them" or "I like both of them."

PQ2: Please leave a comment on your thoughts on tool #1 and tool #2 (e.g., pros and cons):

The question allows the participant to enter a free-form answer.

5.4.6 Replication Package

The source code, code-description mapping database, evaluation output of the tool, and user study data are available on our website[4].

5.5 Evaluation Results

In this section, we show the results from the user study. Our tool generated a total number of 442 comments (maps to 782 code locations) for the 16 Java projects. A breakdown of the number of generated comments is shown in Table 5.5. We also show the number of generated comments from a previous work, CloCom [7], on the same 16 Java projects in Table 5.5 for comparison.

5.5.1 Participant Ratings

We show the human judgment results from the user study in Table 5.6. In the design of our study (Section 5.4.4), we ask participants to write down a comment to describe the given code segment. We collected 240 answers over the 20 participants (12 questions from each group). Participants had given up on writing a comment on 19 (group 1) and 33 (group 2) of the 240 questions because they did not feel that they fully understand the code segment. Amongst all the results, 23 (group 1) and 20 (group 2) of the 240 responses contain a rating of *strongly agree* on all four criteria (completeness, conciseness, expressiveness, and usefulness).

The result shows that CloCom+ is not able to generate high-quality code comments consistently for code segments. Based on the 240 responses, the majority of the participants (more than 50%) strongly agree or weakly agree that the generated comments are complete, concise, expressive and useful in helping them understand the code segments. Compared to our previous work [208], the yield and quality had improved, but it is still not good enough for direct usage.

We summarized the agreement between the participants using a statistical measure called Fleiss' kappa [226]—κ in Table 5.6, which allows us to measure the interrater agreement of three or more raters. A κ value of less than 0 means poor agreement where a value of 1 means perfect agreement. We applied the test to the two groups, where each

[4]http://asset.uwaterloo.ca/AutoComment2/

Table 5.6: Human participant judgments on the generated comments by CloCom+. CP: completeness; CS: conciseness; EP: expressiveness; US: usefulness.

	Human Participant Responses			
Responses (set 1)	**CP**	**CS**	**EP**	**US**
Strongly Agree	37 (30.8%)	69 (57.5%)	53 (44.2%)	47 (39.2%)
Weakly Agree	33 (27.5%)	29 (24.2%)	40 (33.3%)	33 (27.5%)
Disagree	29 (24.2%)	17 (14.2%)	21 (17.5%)	24 (20%)
Strongly Disagree	29 (17.5%)	17 (4.2%)	21 (5%)	24 (13.3%)
Total	120 (100%)	120 (100%)	120 (100%)	120 (100%)
Fleiss' Kappa	-0.01	-0.01	-0.04	-0.01
Responses (set 2)	**CP**	**CS**	**EP**	**US**
Strongly Agree	74 (30.8%)	111 (46.3%)	99 (41.3%)	85 (35.4%)
Weakly Agree	61 (25.4%)	70 (29.2%)	68 (28.3%)	68 (28.3%)
Disagree	68 (28.3%)	39 (16.3%)	53 (22.1%)	61 (25.4%)
Strongly Disagree	37 (15.4%)	20 (8.3%)	20 (8.3%)	26 (10.8%)
Total	120 (100%)	120 (100%)	120 (100%)	120 (100%)
Fleiss' Kappa	0.13	0.08	0.02	0.04

```
1 if ( url == null ) {
2   final ClassLoader standardClassLoader = Thread.
      currentThread().getContextClassLoader();
3   if ( standardClassLoader != null ) {
4    url = standardClassLoader.getResource(
       configurationResourceName );
5   }
6   if ( url == null ) {
```

Figure 5.11: Code segment from Hibernate project for the classloader comment.

group contains 10 participants. The test is done on four categories (i.e., completeness, conciseness, expressiveness and usefulness) on a four-point Likert scale over twelve subjects (questions). In the first group, we obtained κ values of -0.01 for completeness, -0.01 for adequacy, -0.04 for conciseness, and -0.01 for usefulness. The result shows poor agreement overall. In the second group, we obtained κ values of 0.13 for completeness, 0.08 for adequacy, 0.02 for conciseness, and 0.04 for usefulness. The κ values indicate little to no agreement between the participants.

Criterion 1: Completeness

For the completeness criterion, 30.8% and 30.8% of the participants strongly agree that the generated comments are complete from the two groups. The main cause of disagreement is that CloCom+ cannot extract comments that contain correct project-specific information, which usually includes important information. CloCom+ determines the fitness of a sentence based on the posts' score and the calculated text similarity score. While in general using the above metrics are more likely to generate a good code comment, it is not always true. For example, our tool generated the comment, "*Try the current Thread context classloader*" for the code segment from the Hibernate project in Figure 5.11. It selected this sentence because the code segment has text similarity terms, "current," "context," "classload," and "thread" against the automatically generated comment (Section 5.3.5).

The automatically generated a comment that describes the threading code, but it does not describe the semantics of the URL variable in line one, which contains project-specific information. This is a limitation of the crowd-sourced approach for automatic comment generation, where the generated comment either contains project-specific information that is likely to be incorrect or contain no project-specific information.

```
} else {
  File dir = file.getParentFile();
    if (!dir.exists()) {
```

Figure 5.12: Code segment from project Vuze.

```
if (bytesLeft >= length) {
  // we add the data to the end of the buffer
  System.arraycopy(b, offset, buffer, bufferPosition, length
     );
  bufferPosition += length;
```

Figure 5.13: Code segment from project mdrill.

Criterion 2: Conciseness

57.5% and 46.3% of the participants strongly agree that the generated comments are concise from the two groups. The main cause of disagreement on conciseness is that CloCom+ is usually describing generic code segments with generalized text. For example, our tool generated the comment (Stack Overflow post #3693152), "*Sometimes the directory entries aren't already created*" for the code segment from the Vuze project in Figure 5.12.

The code only requires information about creating a file if it does not exist. The problem is that Stack Overflow sentences by default are written for people who want a descriptive explanation about the code's functionality, which is the reason that it can be not concise. In this example, words such as "sometimes" makes the sentence less formal and less concise.

CloCom+ also extracts code comment from existing source code (i.e., GitHub). However, code comments from existing code also can suffer the same issue. For example, our tool generated the comment "*we add the data to the end of the buffer*" for the Vuze project by extracting it from code in the GitHub project, mdrill, in Figure 5.13.

Criterion 3: Expressiveness

44.2% and 41.3% of the participants strongly agree that the generated comments are expressive from the two groups. The main cause of disagreement on expressiveness is due to the automatically generated comment containing wrong information. However, there

```
1 String get_line = lines.get(0);
2 get_line = get_line.substring( get_line.indexOf( ' ' ) + 1 )
     .trim();
3 get_line = get_line.substring( 0, get_line.indexOf( ' ' )).
     trim();
4 int x_pos = get_line.indexOf( '?' );
5 if ( x_pos != -1 ){
6   get_line = get_line.substring( 0, x_pos );
7 }
8 x_pos = get_line.lastIndexOf( '/' );
```

Figure 5.14: Code segment for the comment "*remove query string part.*"

are also some instances where the user interpreted the code in a different way that causes disagreement. For example, CloCom+ generated "*remove query string part*" for line 4—6 for the code segment in Figure 5.14.

One of the participants wrote the comment, "*Get line up to first question mark*" for the code segment and rated the automatically generated comment as strongly not expressive even though both comments are technically correct.

Criterion 4: Usefulness

39.2% and 35.4% of the participants strongly agree that the generated comments are useful to developers from the two groups. The main cause of disagreement on usefulness is that the code is easy-to-understand (so that no comment is needed to help comprehension), or the comment is too trivial. CloCom+ generated the comment, "*Splitting a string at a particular position in java.*" for the code segment (Stack Overflow post #28419586) in Figure 5.15. It is considered not useful by the participants because the code is performing a rather simple operation.

To improve the usefulness of the comments that are generated automatically by CloCom+, we may design code and comment complexity metrics to filter out simple comments and simple code segments in future evaluations. Another way to tackle this issue is to utilize a different data source.

```
1 int c= arg.indexOf(" "); //$NON-NLS-1$
2 int t= arg.indexOf(" ", c+1); //$NON-NLS-1$
3 String className= arg.substring(0, c);
4 String testName= arg.substring(c+1, t);
5 String status= arg.substring(t+1);
6 String testId = className+testName;
```

Figure 5.15: Code segment for the comment "*Splitting a string at a particular position in java.*"

SumSlice Results

We summarized the results of SumSlice in Table 5.7. Do note that we only assigned each participant with three SWUM generated code comment for evaluation (the evaluation of CloCom+ has twelve) to limit the total number of questions to fifteen per participant. The distribution of the responses is slightly different compared to CloCom+. Their technique achieved a much higher completeness rating than CloCom+ because it can generate a natural language sentence that describes the entire method signature (e.g., return variable, method name and input arguments), where participants had also pointed out during the post-study.

The interesting thing to note is the conciseness, which performs worse compared to CloCom+. The reason is that SumSlice generates a second sentence that rates the method's importance based on the PageRank algorithm. For example, SumSlice generated most comments with a similar structure as the following: "*This method gets the full name and returns a String. GetFullName() seems less important than average because it is not called by any methods.* Another possible explanation for the conciseness rating is that the participants are used to the short comments that are generated by CloCom+ during the evaluation, which causes a sudden change in expectation when they encounter comments generated by SumSlice.

Post Study

PQ1 asks the participant's preference on CloCom+ and SumSlice. Five participants like both tools (25%), seven participants like CloCom+ more (35%), five participants like SumSlice more (25%), and three participants don't like either of them (15%).

PQ2 asks for participant's open opinion on the pros and cons of the two tools. We randomly sampled eight participants and displayed their opinion in Table 5.8.

Table 5.7: Human participant judgments on the generated comments by SumSlice. CP: completeness; CS: conciseness; EP: expressiveness; US: usefulness.

	Human Participant Responses			
Responses (Set 2)	**CP**	**CS**	**EP**	**US**
Strongly Agree	22 (73.3%)	6 (20%)	14 (46.7%)	15 (50%)
Weakly Agree	4 (13.3%)	6 (20%)	10 (33.3%)	8 (26.7%)
Disagree	4 (13.3%)	5 (16.7%)	6 (20%)	4 (13.3%)
Strongly Disagree	0 (0%)	13 (43.3%)	0 (0%)	3 (10%)
Total	30 (100%)	30 (100%)	30 (100%)	30 (100%)
Responses (Set 2)	**CP**	**CS**	**EP**	**US**
Strongly Agree	8 (26.7%)	3 (10%)	7 (23.3%)	7 (23.3%)
Weakly Agree	8 (26.7%)	10 (33.3%)	12 (40%)	9 (30%)
Disagree	8 (26.7%)	9 (30%)	7 (23.3%)	8 (26.7%)
Strongly Disagree	6 (20%)	8 (26.7%)	4 (13.3%)	6 (20%)
Total	30 (100%)	30 (100%)	30 (100%)	30 (100%)

An analysis through the post-study comments shows that there are no clear answer as to which tool is better (35% for CloCom+ and 25% for SumSlice). Majority of the participants had stated that CloCom+ has the issue of not generating quality comments consistently (participant 1, 2, 3, 5 and 7). There are also issues where CloCom+ fails to generate a code comment that takes the context into account (participant 8). Several of the participants expressed concerns about SumSlice simply rephrasing the method signature (participant 6, 7 and 8). However, SumSlice is able to generate a much more concise and clean summary in comparison (participant 1, 2, 3, 4 and 5).

5.5.2 Execution Time

We executed CloCom+ on an Intel Core i5-3470 CPU with 16GB of RAM. The database generation (Section 5.3.1) took roughly half an hour to execute. The clone detection tool (Section 5.3.3), NLP (Section 5.3.2), code clone pruning (Section 5.3.4) and comment selection (Section 5.3.5) on each project's execution time is shown in Table 5.5, which took 2,101 minutes in total (between 11—1,100 minutes per project).

Table 5.8: Human participant's opinion on the CloCom+ and SumSlice. PA stands for participant.

PA	CloCom+	SumSlice
P1	Tool 1 considers more answers and ideally chooses the one with the best answer. However it may not be relevant to the code in question.	Tool 2 ensures that documentation is clear and concise, but may be harder to implement.
P2	Tool 1 had a couple of hiccups in it, that didn't make me particularly like it if all of those were from the same tool.	Tool 2 kind of makes it seem like the comments were written by a snarky co-worker. While amusing, it's not super useful.
P3	The possibility of reviweing a Stack Overflow question with a similar issue the comment is refering is the best feature of this tool. Sometimes the comment generated did not add any new information, was practically the same method/english written in english (especially for simple functions), in other cases for these same type of functions the comment added content that wasn't really neccesary due to the simplicity of the function/method	Pros: The comments generated where quite concise, but in most of the cases they really communicate fully the intetion of the method or function they were commenting. I'd prefer a bit larger comment but that will provide a better understanding of the problem
P4	I think that is more descriptive and useful	I think this is more concise and less expressive
P5	It seems to be useful at times and not so much at other times.	I really liked how the comments were very consistent. However, I don't think it is fair to compare the 2 methods since the latter was very easy cases consisting of getter/setter methods while the first method had to solve a lot of harder comments.
P6	Tool 1 seems to produce more accurate and concise comments, however, if it is going to be including stack overflow links in source code comments, the result will be very messy, especially if irrelevant stack overflow links are used.	Tool 2 seems to be a lot more verbose, simply repeating what you already know from the function name. It also includes useless info that may later be irrelevant once those methods are used more.
P7	I think on average this generated more useful comments that explain what a code snippet does in the context of the larger program, although sometimes it does generate useless, redundant information.	This tool doesn't really provide anything useful beyond what's already obvious to the developer.
P8	Inaccurate but sometimes can provide info. Lack of context negatively affects results.	Already had comments. Comments generated seem much more clean and accurate but sometimes just repeat of original comment.

5.6 Qualitative Analysis and Discussion

In this section, we analyze the automatically generated comments by CloCom+ (Section 5.6.1); analyze the comments that are written by developers (Section 5.6.2); analyze the properties of the comments that are written by the participants in the user study (Section 5.6.3), and analyze the impact of each design component on the yield of the generated comments (Section 5.6.4).

5.6.1 Properties of the Automatically Generated Comments

To understand the quality issues of the automatically generated comments, we manually studied the 43 automatically generated comments on a large scale Java project, HSQLDB project, produced by CloCom+.

The first step of the analysis involves the removal of comments that are not describing the target project's source code. These comments are selected by CloCom+ despite being not being related to the target project's code segment due to the comment selection technique, which will enable us to understand how to improve other aspects of a comment given that the comment selection component can select relevant comments. In total, three out of the 43 comments are not describing the target project's code segment. The common reason that CloCom+ selects irrelevant sentence is that comment selection technique simply relies on shared terms between the sentences and code segment as a heuristic for selection. For example, CloCom+ generated "*Skip comments and blank lines*" for Java project because the text similarity component detected a shared term, "line," between the target project's code and the GitHub project CloudStack-archive. The problem is that the extracted comment is explaining the code outside of the matched lines. Even though the target code is reading a file line by line, it is not skipping comments or blank lines.

The second step classifies the remaining 40 sentences that describe the code segment. The process involves iterating through the list of comments and creating a label for each identified issue with the automatically generated comment. We iterated through the labels and identified four major issues. We mapped the issues back to the completeness, conciseness, expressiveness and usefulness criteria as shown in Table 5.9. A comment can be labeled as invalid for multiple reasons. The total number of comments that contains at least one of the four reasons is 14 (35%), where the rest of the comments do not belong to any of the reasons. The result shows similar findings as the user study (Section 5.5), where the presented comment does not contain adequate contextual information on the code segment. For example, a code segment may be reading an audio file, but our tool

generates a generic comment that describes the file reading operation instead of the audio file reading operation. While it is possible to generate comments that do not contain information about the context of the code, the usefulness of such comments may not be useful since it is more generic.

We located five comments that contain a grammar mistake (e.g., "*Are called Maps (see implementations of java.util.Map)*"), and three comments with an incomplete sentence (e.g., "*Always English collation*"), which require future work in the natural language processing component for correction. CloCom+ currently only extracts phrases from a sentence, and it does not attempt to detect incomplete sentences, which can cause issues.

Regarding the size of the generated comments. All comments in HSQLDB project consist of a single sentence, where each sentence contains an average of 10 words.

The automatically generated comments by CloCom+ contains missing and useless information. Since the generated comments are mined from existing natural language sentences, they can be wordy or informally written. The natural language processing technique that is used to refine the English sentences can cause the code comment to be harder to understand. There is also a need for a way to detect if the code is too trivial because not all code would require a code comment.

5.6.2 Properties of Developer-written Comments

To analyze the properties of developer-written comments, we manually inspected seven source code files[5] from a mature Java project—Jajuk. The seven files contain 618 lines of code and 765 lines of comment (inflated due to copyright blocks) as reported using CLOC. We first extract all the comments from the source code file, and then classify the comments by their type. We show the comment type distribution of the seven source code files in Table 5.10. We manually excluded compiler related comments (i.e., $NON-NLS-1$), templated messages (i.e., //nothing to do here), automated block comments that contain no content and copyright comments. We also manually aggregate multiple lines of single line comments that are in the same paragraph as one single line comment.

The results in the comment distribution table show that the majority of the source code comments belong to JavaDoc comments, block comments have a low utilization rate, and single line comments occur occasionally. All of the Java classes and methods contain a JavaDoc comment that is written by the developers.

[5] /jajuk-src-1.10.3/src/main/java/org/qdwizard/

Table 5.9: Reasons that an automatically generated comment is not applicable to the new code segment. The percentage is calculated over the 40 comments that describe the code segment.

Reason	Example	Freq. (%)
1- *Completeness*: The comment **contains missing information** that are critical as a source code comment or **contains useless information** that are not needed. These can be information that are specific to the context of the code.	"*Converts a value to this type.*" is missing information about the type of the object that it is converting towards (HSQLDB—NumberType.java).	6 (15%)
2- *Conciseness*: The comment **is wordy or informally written**.	"*Eg, if you were using Calendar and the current day.*" is an informal sentence that was extracted from Stack Overflow (#14542336).	7 (17.5%)
3- *Expressiveness*: The comment is **hard to read or understand**, or it is expressing the **wrong informatin**.	"*autocommit true should never throw.*", is hard to understand as a sentence (HSQLDB—DatabaseManager.java). It does not express the intent of the code in a natural manner.	7 (17.5%)
4- *Usefulness*: The comment is **not useful or too trivial**. The comment is not needed to understand the code, or it is a direct paraphrase of the code.	"*Sleep for a long time*" is a relatively trivial comment that many participants rated as not useful (HSQLDB—DatabaseManagerSwing.java).	10 (25%)

Table 5.10: Distribution of the developer-written comments in project Jajuk. LCode—lines of code; LCom—lines of comment. LCode and LCom are reported using CLOC. We also show the number of manually identified JavaDoc, single line and block comments.

File (.java)	LCode	LCom	# JavaDoc	# Single Line	# Block
ActionsPanel	105	71	10	3	0
ClearPoint	3	26	1	0	0
Header	60	57	7	0	0
Langpack	39	44	6	2	0
Screen	77	147	23	5	0
ScreenState	50	95	17	0	0
Wizard	284	325	46	7	0
Total	618	765	110	17	0

We also investigated the contents of the 110 JavaDoc and 17 single line developer-written comments through manual inspection. We observe that all comments contain a maximum of two sentences, which means the comments are relatively concise. There are two main types of comments that we observed.

The first type simply provides information about the action or purpose of the source code. Here are some examples of the source code. "*Set the header title text.*" and "*Construct a screen.*" are method-level comments that describe the action of the source code. "*Contains a wizard title, a subtitle used to display the name*" describes the purpose of the source code. "*Screens needing to clear wizard cache ... should implement this interface.*" provides the context as to when to use the Java interface.

The second type simply states the full name of an object in a more meaningful manner, where an object can be a method or variable. Here are some examples of the source code. "*Associated action listener.*" is a paraphrase of type name `ActionListener`. "*UI creation.*" is a paraphrase of the method name `initUI()`. "*Problem panel*" provides the full name of the variable `jlProblem`.

We performed a manual classification and found that 52 of the comments are of type one and 22 of the comments are of type two. A source code comment is not always necessarily meaningful. In the manually inspected source code files, all of the classes and methods contain a JavaDoc comment. We found that nearly all the type two comments are a paraphrase of the source code, For example, "*Can go previous.*" is a paraphrase of

Table 5.11: The number of comments (obtained from AST parser) and the comment ratio in brackets (lines of code per comment) of ten randomly sampled Java projects.

Project	Lines of Code	Block Comments	JavaDoc Comments	Single Line Comments
Eclipse	3,488,835	5,932 (588)	26,263 (132)	30,950 (113)
FreeCol	68,767	82 (839)	11,696 (6)	8,175 (8)
Jajuk	133,637	1,626 (82)	6,291 (21)	7,611 (18)
Hibernate	581,642	8,376 (69)	17,010 (34)	21,029 (28)
Vuze	613,557	5,172 (119)	8,520 (72)	21,176 (29)
HSQLDB	168,239	956 (176)	4,341 (39)	8,913 (19)
Sweet Home 3D	97,210	250 (389)	4,752 (20)	5,625 (17)
FreeMind	68,677	1,603 (43)	2,154 (32)	4,994 (14)
JavaHMO	25,631	399 (64)	131 (196)	1,352 (19)
JBidWatcher	28,357	193 (147)	710 (40)	1,518 (19)

the name `canGoPrevious()`. Although developers of Jajuk had written a formal JavaDoc comment for most of the Java methods, a lot of the comments are a simple paraphrase of the source code.

We also show the comment ratio of 10 randomly selected Java projects in Figure 5.11. The result shows that projects such as FreeMind and Hibernate have a high comment ratio compared to the other projects (e.g., FreeMind has a JavaDoc comment for every 32 lines of code). Projects such as FreeCol and Sweet Home 3D utilizes a high number of JavaDoc comments over block comments, but upon a manual inspection, we observed that many of the JavaDoc comments were generated automatically and contained no comments inside the block.

The study on developer-written comments shows that block comment is rarely used in practice compared to JavaDoc comment and single line comment. The JavaDoc and single line comment both contain a maximum of two sentences. The developer-written comments either states the purpose of the code or provide a rephrase of the code in a more meaningful manner.

Table 5.12: Manual classification of 135 participant-written comments' unique properties. Percentage is calculated over 113 comments because 22 of the comments received no answer.

Property	Example	Total #
Comment is a natural rephrase of the source code (code elements names are broken down correctly)	"get `inSampleSize` based on `outWidth` and `outHeight`" (negative example)	114 (84.4%)
Comment only consists of a single concise sentence.	"Reads in properties from a file"	131 (97%)
Comment starts the sentence with a verb.	"Update the text of icon with the given position"	114 (84.4%)

5.6.3 Properties of Participant-written Comments

We performed a manual inspection of the 135 participant-written comments (15 questions over nine randomly selected participants) in the evaluation of positive properties of participant-written comments. The process involves iterating through the list of comments and creating a label for each identified positive property with the written comment. Once we have a list of labels, we attempt to merge the labels that contain overlapping properties. For example, we merged "Comment does not contain directly copied code artifacts (e.g., variable name and method name) in the comment" into "Comment is a natural rephrase of the source code (code elements names are broken down correctly)." We repeat this process until a non-overlapping list of label is discovered.

We identified three labels that may guide future techniques to improve automated comment generation under Table 5.12.

First, only in rare cases that our participants included the name of the code artifacts (e.g., variable names and method names) in their code comment. For example, a participant wrote, "*FileReader is closed in the try segment, if some exception arises there is a catch segment defined for the same*," which contains the class name, `FileReader`, in the comment. In the majority of the cases, participants will break down the variable into smaller parts. For example, another participant wrote the following comment for the same code segment, "*Cleanup file reader.*" Our tool generated the following comment, "*try to close the reader.*" In the future, we may supplement automatically generated comments with textual information (variable names and method names) from the source code ([90]). The inferencing of contextual information remains a challenging task, which may require

information from the control or data path of the program to inference information.

Second, participant-written comments are in general very concise, where 97% of the comments consist of a single concise sentence.

Third, most of the sentences start with a verb phrase (e.g., "*Get ...*" and "*Create ...*") followed by a noun phrase (e.g., "*the bounds of ...*" and "*a StringWriter ...*"). Such pattern seems to be a standard convention that is consistently utilized by most participants. We also note that the Java language specification ([227]) suggests that method names should be verbs or verb phrases. Previous work ([90]) relies on detecting verbs in the source code to generate comments.

The study on participant-written comments shows three main properties. First, the comments are often a natural rephrase of the source code with a better explanation for the names of the identifiers. Second, almost all comments are concise containing only a single sentence. Third, the majority of the sentences starts with a verb.

5.6.4 Yield Analysis

Since each proposed component has an impact on the number of the generated comments, we conducted an analysis of each component using the Java project—GanttProject to illustrate their impact on the yield.

To start the comment generation, CloCom+ first requires a database. Since GanttProject is a Java project, CloCom+ extracted from Stack Overflow using the `java` tag and extracted 309,593 code-description mappings. The next step involves performing natural language analysis on the extracted mappings to filter out invalid sentences (Section 5.3.2).

The next step performs code clone detection (Section 5.3.3) between the database and input source code from GanttProject (62 KLOC). After applying the heuristic, line percentage matching, text similarity on the detected clones, it reduces the number of code clone match groups to 93. A clone group represents a group of code segments that are similar to each other, which contains one or more code segment from the target project and one or more code segment from Stack Overflow. However, not all clone groups contain a comment candidate. Amongst the 93 clone groups, only 8 contains a comment candidate.

Disabling percentage matching pruning technique introduces 8 new comments on top of the 8 existing clone groups, bringing the total number of clone groups to 14 (do note that a new comment does not necessarily introduce a new clone group). Upon manual inspection, amongst the 8 new unique comments, 6 of which are not describing the target code segment correctly.

```
for (int i = 1; i < 5; i++) {
   ...
}
```

```
int i = 1;
while (i < 5) {
   ...
   i++;
}
```

Figure 5.16: Two pieces of code segment that contain the same computation but are expressed with a different set of syntax.

Disabling repetitive clone pruning heuristic does not impact the yield. It means that the code clone did not match low-level code between the target project against the database.

Disabling text similarity does not impact the yield because all the code comments already contain similar terms against the target source code.

The yield is controlled mostly by the parameters configured on the code clone detection tool and text similarity component, In the code clone detection tool, the minimum clone size threshold (four statements) and gap size (two) directly affects the amount of content in the matched code segments. In the text similarity component, the minimum text similarity term threshold impacts the similarity between the selected comment and the matched code segment. We did not observe any components or heuristics that can be removed to simplify the design.

A potential factor for the low yield comes from the code clone detection tool. The code clone detection tool may not be able to detect more complicated types of code clones. For example, the statement, `c = xx(yy(b));` can be expressed with a different syntax, `a = yy(b); c = xx(a);`. Such type of code clones cannot be detected by the current code clone detection tool. Another example would be a loop that is expressed with two different syntaxes as shown in Figure 5.16.

5.6.5 Limitations

We discuss the major types of general limitations that can affect the practical application of the technique.

First, the results (Section 5.5) show that only 21 of the 105 responses strongly agreed or agreed that the generated comments are both accurate, adequate, concise and useful at the same time. Although most heuristics can be tuned to reduce the number of false positives, they often would further reduce the yield. For example, the natural language processing technique that we use to extract the core part of a sentence is very reliable, but it is clear that more sophisticated natural language processing techniques are needed to understand the semantic meaning of the sentence for a more consistent result.

Second, the yield issue remains an issue with the technique. Our recent work, CloCom [7], attempted to tackle this issue by mining existing software repositories such as GitHub for automatic comment generation. It is interesting to see that CloCom also suffers from the yield issue despite having a much larger code base. Based on the same 15 evaluated Java projects as previous work [90] which do not include Eclipse and Android projects, CloCom generated comments for 197 clone groups whereas CloCom+ generated comments for 100 clone groups. Additionally, previous work [228] had shown that API usage patterns do exist and can be mined from the source code. Therefore, we believe that common code patterns do exist, but these patterns do not necessarily contain a source code comment.

5.7 Threats to Validity

Several threats limit the validity of our experiments. We now discuss these potential threats and how we control or mitigate them.

5.7.1 External Validity

The external validity of the study relates to the extent that we can generalize its results. CloCom+ mines Q&A sites for generating the code-description mapping database. If the Q&A site does not discuss a code segment, then CloCom+ cannot generate a comment for it. It is a limitation that impacts the generalizability, which impacts the yield (number of generated comments). It may be possible to combine our approach with previous techniques ([89, 90]) to determine the important terms in a code segment for the comment selection process.

The current technique only generated 442 comments for the 16 evaluated projects, which means the yield is still rather low. It can limit the generalizability of our technique.

In the future, it is possible to expand the natural language analysis to analyze different parts of a Stack Overflow post to help improve the yield.

In our user study, we randomly sampled code segments during the generation of each questionnaire (Section 5.4.3). We generated a total number of seven questionnaires, where the same questionnaire had been evaluated by every participant in RQ1 of the user study. The advantage is that we can obtain multiple opinions over the same set of questions. The problem with this approach is that we can only cover a subset of the generated comments. Our previous work ([208]) randomly samples code segments from the question set until all code segments had been evaluated, which means each participant will receive a different set of question. Since human opinion is subjective, we chose to have multiple participants to evaluate the same set of code segments for a higher confidence in the results.

5.7.2 Internal Validity

The internal validity of the study is the extent to which a treatment affects a change in the dependent variables. The current code clone detection tool is only capable of detecting code clones that are semantically different if the difference is due to statement reordering or deletion. Since we had implemented heuristics to make the Stack Overflow code segments compilable (Section 5.3.3) to extract the AST, it is now possible to apply other AST-based code clone detection tools to detect more advanced types of code clones.

CloCom+ applied an off-the-shelf natural language processing model to the software domain, which can affect the correctness of the natural language processing technique. Most natural language models, including the Stanford CoreNLP parser, can fall short on interpreting technical terms in the software domain. The reason is that the natural language model was trained on well-written text such as the Wall Street Journal. For example, "file directory" is a noun phrase in the software domain, but most natural language parsers would treat it as a verb phrase. The Stanford parser that we used in this work has worked well in our experiments, which was able to generate a correct parse tree by analyzing the entire sentence. For example, a full sentence, "*Open the file directory*," is reliably interpreted by the parser to contain a verb followed by a noun phrase. For sentences in Stack Overflow posts that are not full sentences, we mitigated this issue by applying the main subtree extraction technique to filter them out.

In RQ2 of the user study, each question contains two comments, including a participant-written comment and an automatically-generated comment. The participants were unaware of the source of each comment, but when we generate the questionnaire, the order of the two comments is always the same. The participant-written comment is always labeled

as the first comment, and the automatically-generated comment is always labeled as the second comment. The ordering might cause a learning threat to our user study.

The subjects of our user study consist of graduate and undergraduate students, which might not be representative of real developers. Also, subjects have different programming language experience. Therefore, their judgment could differ based on their programming language experience. The subjects in our study have an average of 5.4 years of programming experience (between 3-9 years), and all participants have industry programming experience.

The qualitative study contains multiple manual evaluations of the source code segments and comments. Our opinion might not be representative of real developers, but we tried our best to be objective during the evaluation. In the evaluation of automatically generated comments (Section 5.6.1), we extracted the classification labels from the user study results (Section 5.5) to ensure the labels are not just the author's opinion. In developer-written comments (Section 5.6.2), we added automated analysis on the code comment ratio to support the manual analysis.

5.7.3 Construct Validity

Construct validity concerns the relation between theory and observations. In our evaluation, we proposed metrics to evaluate the quality of the comments based on the completeness, conciseness, expressiveness and usefulness criteria. A potential threat from the criteria is that it does not directly indicate the quality of the comment. For example, a comment can be accurate, adequate, concise but not useful, which makes it unclear as to whether the comment is still applicable to the source code. In our evaluation, we mitigate this threat by showing that 21 out of the 105 responses satisfied all four criteria with a rating of strongly agree or agree. Previous work by [89] used a similar set of metrics, with the difference that they did not evaluate the comments using the usefulness criterion. Therefore, we believe that these metrics are valid for evaluation. Also, several parts of the qualitative study rely on the observations from the authors, which may not be representative of the actual theory.

The usefulness criterion may not be a practical measure on the generated comments because the usefulness of a comment depends on the context of the code, background of the participant, and the summarization task that is given to the participant. Previous work ([229]) performed a pilot study on how a target task can impact the content of the software summary. In their study, they considered two summarization tasks, a summary directed towards the code tester and another summary directed towards the code reuse. The results show that when a specific target task is given to the user, the summaries

would contain words that are specific towards testing and words that are specific towards software reuse. They also show that summarization of unfamiliar code is challenging to the participants and having dynamic information related to the code would be useful (e.g., GUI code can have the running GUI shown to the user). We attempted to mitigate this threat by training the participants to understand that the code summaries are code comments that a programmer will write for other developers to read (Section 5.4.4).

5.7.4 Conclusion Validity

Conclusion validity threats deal with the relation between the treatment for/on the outcome. We only concluded that our tool works Java and Android projects, but our technique should apply to other similar compiled languages such as C and C# because the proposed techniques and tools are language independent. CloCom+ is designed to extract code comments from a particular data source, Stack Overflow, which may impact the design thresholds that are utilized in CloCom+.

5.8 Summary

We presented an approach for automatic comment generation using Q&A sites. Our approach leverages natural language processing, code clone detection, and text similarity analysis techniques to map a natural language sentence into a source code comment.

A user study with 20 participants evaluated the 442 comments that are generated by CloCom+. The number of generated comments from CloCom+ is still rather low and the quality still requires improvement. Future possible paths include the extension of our tool to analyze other parts of a sentence or use more advanced code clone detection tools that can detect code insertion and reordering. Our user study shows that the majority of the participants consider the automatically generated comments to be complete, concise, expressive and useful. However, the statistical tests had shown that there is little agreement between the participants. We compared our work against previous work, SumSlice. The result shows that their work can generate more complete summary compared to CloCom+, but the generated summary is not very concise.

Chapter 6

Future Work

In the future, it may be possible to combine two proposed branches of work (documentation analysis and documentation generation) to further improve software dependability. However, automated documentation generation is still a rather new area of research that requires more time to mature. Our proposed technique on automated comment generation (CloCom+) still needs further work to improve the completeness and usefulness of the generated comments. Also, CloCom+ currently generates code comments to describe the general source code, which is not applicable to DASE which relies on analyzing code comments from the header files that describes the data structure.

We present additional work to complete my thesis, which expands upon other ways to automate the constraint extraction process. We propose an approach to automatically extract file format constraints from documentation, and discuss on how to potentially apply the extracted file format constraints to a file parser.

6.1 Detecting Bugs using Documentation Constraints

Applications that accept input files of a specific structure are called structured-file parsing applications. Some file formats contain a rigid structure, where each data field can be directly accessed at a defined offset (e.g., ELF, PNG, JPEG, and mp3). There are also file formats that contain a loose structure (e.g., PDF, XML and programming source code), which requires a lexer to tokenize the input, and a parser to analyze the tokens based on the grammar of the input. Our previous work, DASE [133], analyzed the ELF file format, which has a mostly rigid structure as shown in Figure 3.4. The ELF header, section header

```
typedef struct {
   unsigned char e_ident[EI_NIDENT];
   uint16_t      e_type;
   uint16_t      e_machine;
   uint32_t      e_version;
   ElfN_Addr     e_entry;
   ElfN_Off      e_phoff;
   ElfN_Off      e_shoff;
   uint32_t      e_flags;
   uint16_t      e_ehsize;
   uint16_t      e_phentsize;
   uint16_t      e_phnum;
   uint16_t      e_shentsize;
   uint16_t      e_shnum;
   uint16_t      e_shstrndx;
} ElfN_Ehdr;
```

Figure 6.1: Data field layout details of the ELF Header.

table, and program header table each contain a list of positional offsets for each data field, which makes it a rigid structure. Figure 6.1 shows the struct definition of the file header, which defines the positional offset of each field. For example, the header starts with a char byte (`e_ident`) that describes how to interpret the file, followed by an unsigned int (`e_type`) that indicates the object file type. Structured-file parsing applications are difficult to test because they contain a large number of execution paths to test.

Existing symbolic execution testing tools such as KLEE [40] do not scale on structured-file parsing applications because the lexer and parser of a program often generate a large number of execution paths. Suppose we have a program that processes a string input from a file (e.g., "`int i = 1;`"), where the string input does not have a rigid structure. The token separator between "int" and "i" may be replaced by one or more white space, line break, or tab character. Each possible combination of the token separator generates a new execution path, and the separator can have any possible length. Without an understanding of the input file's structure or grammar, given a symbolic input, the symbolic execution engine may fork many execution states for each possible combination of the token separator, which leads to the path explosion problem. Our previous work, DASE [133], attempted to bound the input of a program with file format constraints. However, due to limitations on the expressiveness of the constraint system, DASE can only specify constraints on file formats that have a rigid structure. For example, suppose we want to constraint the following dictionary string from a PDF file that defines a list of key-value pairs, where there are two

```
<<
/Type /Catalog <- order of the key/value fields cannot be assumed
/Pages  2 0   R  <- extra token spacing that affects the byte offset
>>
```

Figure 6.2: An example of where the constraint cannot be applied directly on a static byte offset due to the extra white space between character '0' and 'R.'

keys, `Type` and `Pages`, that contain the value, `/Catalog` and `2 0 R`, respectively.

```
<</Type /Catalog /Pages 2 0 R>>
```

KLEE [40] and DASE [133] are not scalable to programs that process the above type of input because the lexer and parser of the application will cause the symbolic execution engine to keep forking branches that contain an invalid input. While it is possible to constrain the dictionary string by assuming the position of the key/value fields using DASE, it also prunes away many of the other valid input forms such as the string below, where the content of the dictionary string is reversed, and the spacing is different. Figure 6.2 shows an example where the positional offset of the data inside a PDF file cannot be assumed due to two reasons. First, we cannot assume the order of the two keys, 'Pages' and 'Type,' to always be the same. Second, there may be variations on the syntax of the data (e.g., token spacing). Both issues can only be addressed with a parser that understands the file format.

If we configure DASE to assume that the first value, `/Catalog`, will always start at byte position 9, such constraint will be invalid if applied to the following string.

Therefore, we propose to address the challenges of modeling structured-file inputs to help improve symbolic execution. We believe the utilization of a string solver is a promising direction to model the input file format to avoid forking execution states that contain an invalid input. Figure 6.3 shows the context-free grammar that represents the valid structure of the dictionary string, which can be converted into a constraint using existing string solvers such as HAMPI [230].

The grammar allows the symbolic execution tool to be aware of the following list of symbolic tokens and the collection of high-level constraints (as opposed to the traditional byte-level constraints) during symbolic execution.

1. «

```
dictStr := "<<" keyValuePair + ">>";
keyValuePair := "/" + keyName + space + value | keyValuePair +
space + keyValuePair;
value := keyName | reference;
reference := numbers + space + numbers + space +  "R";
keyName := "/" + (letter)+;
letter := [a-z];
numbers := (number)+;
number := [0-9];
space := ("\s"|"\n")+;
```

Figure 6.3: Context-free grammar that describes the valid structure of the dictionary string.

2. keyName
3. value
4. keyName
5. value
6. »

We discuss two possible directions for exploration. The first direction is to explore the combination of symbolic execution with fuzzing [124, 126, 13]. Previous work [13] had shown that it is possible to leverage grammar-based constraints to help generate well-formed inputs for fuzzy testing. Their technique first utilizes a lexer to tokenize the input, which allows the utilization of grammar-based constraints for the specification of high-level constraints. We would like to explore the extraction of grammar-based constraints automatically from documentation to model file formats.

The second direction is the learning of file formats automatically. For example, the dictionary string discussed above contains a key, `Pages`, that connects to object number two. It may be possible to capture this semantic relation between objects through analysis on the collected constraints during symbolic execution. One potential downside of this approach is that the learned constraints will depend on the correctness of the input files.

Challenges The challenge comes from the automatic extraction of the file format constraints from the documentation. We require file format constraints that describe the lexical (i.e., tokens) and grammatical part of the file format, which differs from our previous work in DASE [133] where we focused on extracting value constraints. We describe two approaches to automatically extract constraints for the PDF file format [166], and an approach to apply the constraints on a PDF parser.

6.1.1 Automated Constraint Extraction using Regular Expressions

Since documentations contain constraints that describes the file format of structured file formats, we demonstrate an approach to extract file format constraints automatically from the ISO documentation of the PDF file format [166].

There are many types constraints in a documentation that describe a PDF file format, such as the general layout of the file format, specification of the supported data types (e.g., integer numbers, real numbers, strings and booleans), special characters (e.g., list of valid whitespace characters and delimiters) and dictionary constraints (e.g., formatting of the key-value pairs). We implemented a constraint extractor to extract *dictionary constraints* because dictionary constraints help validation on the data's correctness, which is important for file recovery because it prevents the parser from extracting incorrect data.

Extracting Dictionary Constraints

A dictionary is an associative array data structure. It consists of a list of key-value pairs, where each key must only appear at most once in the collection. Dictionary constraints are documented inside tables in the PDF documentation. Since all the tables in the documentation follow a consistent format, we utilize regular expressions to extract the constraints.

The PDF file format contains many types of dictionaries (e.g., `trailer`, `pages`, `page`, and `root`). Each dictionary contains key-value pair entries that define the properties of an object. Dictionary constraints can be used to understand each key-value pair's expected format and value. Below is an example of a `trailer` dictionary, which contains two key-value pairs, "Root" and "Size."

```
1 <</Root 1 0 R /Size 5 >>
```

This `trailer` dictionary contains a list of valid key-value entries which are specified in the documentation (Table 6.1). We extract the following constraints for the key, "Root," which is defined for the `trailer` dictionary in the documentation:

- Value format: `dictionary`
- Is it a mandatory key: `yes`
- Is it inheritable: `not defined`

```
(entries\sin\s(a|an|the)|
entries\scommon\sto\sall|
entries\sspecific\sto\sa|
required\sentries\sin\sa)
\s(.+)\s
(dictionary|object|stream|root|node|[a-zA-Z]+)
```

Figure 6.4: The first regular expression for matching table captions.

- Does it have to be an indirect reference: `yes`
- Dictionary type: `catalog`
- Dictionary location: `Section 7.7.2 in the documentation [166]`

The key, `Root`, contains the value, `1 0 R`. The extracted constraints allow us to validate the correctness of the value. The constraint stated that the value has to be an indirect reference, which means it has to be written in the format of two numbers followed by the keyword "R," where the formatting of an indirect reference is defined in a different part of the documentation.

We automatically parsed 137 tables that contain constraints related to the dictionaries from the PDF documentation by defining two regular expressions. The two regular expressions are written manually after we identified the patterns that are common between the tables in the documentation.

The first regular expression, presented in Figure 6.4 ("\s" refers to spaces), identifies tables that are related to dictionary parsing by matching against table captions.

The table caption for Table 6.1, "*Table 15 - Entries in the file trailer dictionary.*", satisfies regular expression #1. We extract match groups from the regular expression to identify properties of the table. For example, the first match group (first four lines) matches "*Entries in the*". The second and third match group identifies the type of the dictionary (e.g., pages object and stream dictionary), which is "file trailer" in this example.

The second regular expression, shown in Figure 6.5, decompiles the table entries of Table 6.1 into three parts, including a "Key," "Type" and "Value" table entry. For example, Table 6.1 has an entry where the "Key" is "Root," "Type" is "dictionary," and "Value" is *"(Required; shall be an ...)"*.

```
([a-zA-Z]+)\s
(integer|boolean|dictionary|
array|name|stream|number|
real|string|text\sstring|
rectangle|date|name\stree|
[a-z]\sor\s[a-z])
\s(.+)
```

Figure 6.5: The second regular expression for decompiling table entries.

Table 6.1: Trailer dictionary's table from the ISO 32000 documentation specification. The trailer dictionary's table contains the caption, "Table 15 - Entries in the file trailer dictionary."

Key	**Type**	**Value**
Prev	integer	(Present only if the file has more than one cross-reference section; shall be an indirect reference) The byte offset in the decoded stream from the beginning of the file to the beginning of the previous cross-reference section.
Root	dictionary	(Required; shall be an indirect reference) The catalog dictionary for the PDF document contained in the file (see 7.7.2, "Document Catalog").
Encrypt	dictionary	(Required if document is encrypted; PDF 1.1) The document's encryption dictionary (see 7.6, "Encryption").
...	...	...

6.1.2 Automated Constraint Extraction using Natural Language Analysis

The PDF file format supports eight basic object formats, including boolean values, integer/real numbers, strings, names, arrays, dictionaries, streams, and the null object. The descriptions of the object formats are written in English in a free-form instead of a structured table format. Thus, it is ineffective to process them with regular expressions. Therefore, we adapt and apply natural language processing techniques to process them. The PDF documentation contains dedicated chapters that describe each object format. We manually identified the chapters that describe the object format constraints (chapters 7.3.2 to 7.3.9) and supplied all the English sentences to the constraint extraction tool.

Table 6.2: Syntactic categories examples for the production rules in the CFG.

Symbol	**Meaning**	**Example**
S	sentence	an integer shall be written as one or more decimal digits optionally preceded by a sign
PP	prepositional phrase	as one or more
VP	verb phrase	shall be written as one of the following two ways
SBAR	subordinating conjunction clause	when writing an name in a PDF file
REQUIREMENT	requirement	one of the following two ways
ACTION	action	writing a name in a PDF file
...	...	...

Object format constraint limits an object's appearance. For example, an `integer number` object has to contain one or more decimal digits optionally preceded by a sign (e.g., "`+17`"). Since the dictionary constraints (Section 6.1.1) specify the expected object format for each key-value pair, we can use object format constraints to validate if an object conforms to the format. For example, Table 6.1 contains the key, "Prev," that requires an "integer" object format. Thus, we can use the integer object format constraints to validate if a "Prev" value is an integer object.

To parse the English sentences, we defined a domain-specific context-free grammar (CFG) to extract object format constraints (Figure 6.7). The purpose is to use formal grammar to describe and label the high-level parts of a sentence. The CFG in Figure 6.7 contains a *starting symbol*, 'S,' that represents the input sentence. Each *production rule* contains a non-terminal symbol (e.g., PP, VP, NP, SBAR, etc.) that can be replaced by a pattern consisted of terminals and non-terminals. The syntactic category of each symbol is shown in Table 6.2. Terminals are words in a sentence. For example, the non-terminal symbol, 'DT' (determiner), can be represented by two words, 'a' or 'an'.

Consider the sentence from the PDF documentation that describes the integer object, "*An integer shall be written as one or more decimal digits optionally preceded by a sign.*" The production rule, `'REQUIREMENT,'` defines the syntactic structure of the string, "*shall be written as one or more decimal digits optionally preceded by a sign.*"

We defined a CFG to focus the parsing on three different categories of sentence structures (production rule 'S' in Figure 6.7 contains three structures). The first category starts

with a noun phrase (NP) followed by a verb phrase (VP). The second category is a prepositional phrase (PP). The third category starts with a subordinating conjunction clause (SBAR) followed by a noun phrase (NP) and a verb phrase (VP). Our approach is similar to a natural language parser, but our approach includes custom part-of-speech tags (e.g., REQUIREMENT, ACTION, APPEARANCE) for the production rules. The reason is that we wanted to label specific high-level phrases such as the requirement phrases (e.g., "*shall consist of ...*") or format phrases (e.g., "*a sequence of ...*"), and be able to retrieve them as a constraint later on.

The output of the parser is the parse tree of the sentence. Figure 6.6 shows the labeled parse tree of the following sentence: *"An integer shall be written as one or more decimal digits optionally preceded by a sign."* The nodes of the tree represent the terminals in the grammar, and the leaf nodes represent the non-terminals (words) in the sentence. For example, we can extract from the sentence that the object ("*an integer*") contains the requirement ("*shall be written as ...*"). The requirement contains a format specification ("*one or more decimal digits ...*").

After obtaining the parse tree of a sentence, we applied three templates to extract constraint from the parse trees:

The first template focuses on sentences that describe an object's format. It detects such sentences by checking for the existence of a 'REQUIREMENT' and 'FORMAT' node. It extracts the value of specific nodes including 'VALUE,' 'OBJECT,' 'FORMAT,' and 'KEYWORD,' and returns either 1) a string that can be a keyword or regex or 2) a list that contains the identified tokens. Based on the previous example, *"An integer shall be written as one or more decimal digits optionally preceded by a sign."* contains the 'REQUIREMENT' node, '*shall be written as xxx*,' and the 'FORMAT' node, '*one or more decimal xxx.*' It statically maps the format string into a regular expression. The template can also return a list. For example, *"A stream shall consist of a dictionary followed by zero or more bytes bracketed between the keywords stream and endstream."* contains a 'REQUIREMENT' node and 'FORMAT' node. The template extracts the 'KEYWORD' nodes that exist within the 'FORMAT' node and returns a list containing two keywords, `stream`, and `endstream`.

The second template focuses on sentences that describe the valid enclosure format of an object. It detects such sentences by checking for the existence of a 'FORMAT' and 'PP' node. Once detected, it extracts the value of the 'ENCLOSE' node and returns a list of valid tokens. For example, *"As a sequence of literal characters enclosed in parentheses ( )."* will return a list containing two characters, '(' and ')'.

The third template focuses on sentences that describe how an object should appear in

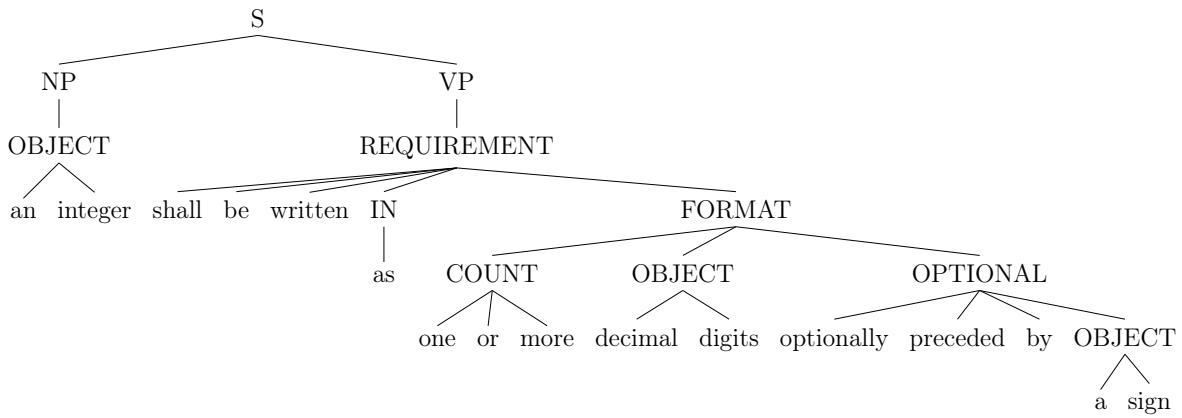


Figure 6.6: Parse tree for the sentence: "*An integer shall be written as one or more decimal digits optionally preceded by a sign.*"

a PDF file. It detects such sentences by checking for the existence of an 'APPEARANCE' and 'REQUIREMENT' node. Once detected, it extracts the value of the 'FORMAT' node and returns a list of tokens. *"They appear in PDF files using the keywords true and false."* will return a list containing two keywords, `true` and `false`.

6.1.3 Applying File Format Constraints to a File Parser

We build a constraint-based parser that utilizes file format constraints (dictionary constraints) to (1) enable fault-tolerant parsing of the data and (2) validate the semantic correctness of the parsed data. A parser is critical towards the recovery process since it is responsible for extracting data from the objects. Existing recovery tools such as mutool [73] Poppler [77] do not utilize dictionary constraints to validate the key-value pairs in their PDF parser. Previous work by Endignoux et al. performs constraint validation on dictionary constraints [70]. However, our proposed approach differs in two ways. First, their algorithm rejects the entire PDF file upon detecting constraint violations, whereas our proposed approach discards the part that violates the constraint to allow a parser to recover from the error. Second, their constraints are manually written whereas our proposed approach extracts them automatically from the documentation.

Th following shows an example where a constraint-based parser can identify and avoid the parsing of invalid values, where the parser's job is to return a list of key-value pairs:

⟨*S*⟩ ::= ⟨*NP*⟩ ⟨*VP*⟩ ⟨*ENDING*⟩
| ⟨*PP*⟩ ⟨*ENDING*⟩
| ⟨*SBAR*⟩ ‘,’ ⟨*REQUIREMENT*⟩ ⟨*ENDING*⟩

⟨*PP*⟩ ::= ‘as’ ⟨*FORMAT*⟩

⟨*VP*⟩ ::= ⟨*REQUIREMENT*⟩
| ⟨*ACTION*⟩

⟨*NP*⟩ ::= ⟨*OBJECT*⟩

⟨*SBAR*⟩ ::= ‘when’ ⟨*VP*⟩

⟨*MULTI-OBJECT*⟩ ::= ⟨*DT*⟩ ⟨*POSITION*⟩ ‘,’ ⟨*POSITION*⟩ ‘,’ ‘or’ ⟨*POSITION*⟩ ⟨*OBJECT*⟩
| ⟨*OBJECT*⟩ ‘and’ ⟨*OBJECT*⟩

⟨*REQUIREMENT*⟩ ::= ‘shall’ ‘be’ ‘written’ ⟨*IN*⟩ ⟨*FORMAT*⟩
| ‘shall’ ‘consist’ ‘of’ ⟨*OBJECT*⟩ ⟨*FORMAT*⟩
| ‘shall’ ‘be’ ‘only’ ⟨*NUM*⟩ ⟨*OBJECT*⟩ ‘,’ ‘denoted’ ‘by’ ⟨*OBJECT*⟩
| ⟨*FORMAT*⟩
| ⟨*APPEARANCE*⟩

⟨*APPEARANCE*⟩ ::= ‘appear’ ‘in’ ⟨*OBJECT*⟩ ⟨*FORMAT*⟩

⟨*ACTION*⟩ ::= ‘writing’ ⟨*OBJECT*⟩ ⟨*IN*⟩ ⟨*OBJECT*⟩

⟨*FORMAT*⟩ ::= ⟨*COUNT*⟩ ⟨*OBJECT*⟩ ‘with’ ⟨*OPTIONAL*⟩
| ‘one’ ‘of’ ‘the’ ‘following’ ‘two’ ‘ways’
| ⟨*COUNT*⟩ ⟨*OBJECT*⟩ ⟨*OPTIONAL*⟩
| ‘a’ ‘sequence’ ‘of’ ⟨*OBJECT*⟩ ⟨*ENCLOSE*⟩
| ⟨*OBJECT*⟩ ⟨*ENCLOSE*⟩
| ‘followed’ ‘by’ ⟨*COUNT*⟩ ⟨*OBJECT*⟩ ⟨*ENCLOSE*⟩
| ⟨*KEYWORD*⟩ ‘shall’ ‘be’ ‘used’ ‘to’ ‘introduce’ ⟨*OBJECT*⟩
| ‘using’ ⟨*KEYWORD*⟩

⟨*KEYWORD*⟩ ::= ‘the’ ‘keywords’ ⟨*VALUE*⟩ ‘and’ ⟨*VALUE*⟩
| ⟨*DT*⟩ ‘solidus’

⟨*ENCLOSE*⟩ ::= ‘enclosed’ ‘in’ ⟨*OBJECT*⟩
| ‘bracketed’ ‘between’ ⟨*KEYWORD*⟩

⟨*OPTIONAL*⟩ ::= ‘optionally’ ‘preceded’ ‘by’ ⟨*OBJECT*⟩
| ⟨*DT*⟩ ‘optional’ ⟨*OBJECT*⟩ ‘and’ ⟨*MULTI-OBJECT*⟩

⟨*COUNT*⟩ ::= ‘one’ ‘or’ ‘more’ | ‘zero’ ‘or’ ‘more’

⟨*ENDING*⟩ ::= ‘.’ | ‘:’ | ‘;’

⟨*VALUE*⟩ ::= ‘true’ | ‘false’ | ‘stream’
| ‘endstream’

⟨*NUM*⟩ ::= ‘one’

⟨*POSITION*⟩ ::= ‘leading’ | ‘trailing’ | ‘embedded’

⟨*IN*⟩ ::= ‘as’ | ‘in’

⟨*DT*⟩ ::= ‘a’ | ‘an’

Figure 6.7: CFG for extracting constraints from natural language sentences. One of the production rules, OBJECT, is omitted since it contains a full list of object related english terms.

```
1 trailer
2 <</Root 1 0 R /Size 5>>
3 endobj
```

Listing 6.1: Example of a constraint-based parser at identifying invalid values

- Root, 1 0 R
- Size, 5

Existing recovery tools [73, 77] assumes a value must follow a key, but they do not validate the format of the value. Suppose the value of the key, `Root`, becomes a numeric value, '`1`', instead of an indirect reference, "`1 0 R`". Since mutool does not know the expected format of the value, it will accept '1' as the value for `Root` if "`0 R`" is corrupted or missing. The acceptance of values that violate the constraints causes mutool to fail to generate a correct PDF file. For example, mutool generates the following trailer fix because it does not validate the correctness of the key-value pairs:

```
trailer
<</Size 5/Root 1>>
```

On the other hand, our constraint-based parser recognizes the violation of the constraint and rejects the value, which forces the repair operator to generate a new trailer as shown below. Do note that the value of "size" had increased because the repair operators added new objects to the PDF file during the repair process.

```
trailer
<</Root 6 0 R /Size 7>>
```

Existing recovery tools attempt to detect invalid syntax during the parsing process, but they do not validate the syntax during the parsing stage. The above trailer example shows that such basic heuristic is not sufficient. For example, the reason that the value '1' is accepted as a valid value by their parser is that it conforms to the syntax of a numeric object. Although the syntax is correct, it is not semantically correct which prevented the repair operators from triggering to repair the incorrect file trailer.

Another advantage of validating the semantics of the parsed data is it allows fault-tolerant parsing of the data. Existing parsers attempts to recover from parsing error by

skipping over tokens (from the tokenizer). They generally give up after a certain number of attempts due to the possibility of introducing parsing error. A constraint-based parser reduces the chances of a parsing error since it validates all the key-value pairs within each dictionary. Therefore, our proposed constraint-based parser does not have a limit on parsing attempts.

In order for a constraint-based parser to validate an object, it must know the type of the object it is dealing with. We can achieve this by analyzing the file format's tree structure starting from the root of the document tree (Figure 4.2), and recursively parses the child objects in the tree with guidance from the constraints. For example, the trailer dictionary contains the key, "Root," and the constraints of the child object is documented in chapter 7.7.2 of the documentation (Table 6.1). In the case where it fails to locate the dictionary constraint, it attempts to guess the type of the object by locating the dictionary constraint table that can parse the highest number of key-value pairs.

We implemented the backend tokenizer and parser based on mutool's [73] implementation by mapping the C code into Python. While mutool's tokenizer and parser validate the syntactical correctness of the data, our proposed technique utilizes constraints to validate the semantical correctness of the data.

Chapter 7

Conclusion

This thesis demonstrated several techniques to utilize documentation to improve software dependability. Specifically, we improve both a system's reliability (e.g., failure-free operation) and maintainability (e.g., ease of understanding) using documentation. We propose and implement three pieces of work to support the claim.

Chapter 3 focuses on improving software reliability. The proposed technique, DASE, automatically extracts and applies input constraints from documentation to improve a symbolic execution tool for automated bug detection and test generation.

Chapter 4 focuses on improving software reliability. The work contains an empirical study to study and repair corrupted PDF files. The proposed technique, DocRepair, utilizes manually extracted constraints from documentation to repair corrupted files automatically.

Chapter 5 focuses on improving software maintainability. The proposed technique, CloCom+, improves software documentation by generating source code comments automatically. CloCom+ generates code comments by mining existing software repositories in GitHub and a Question and Answer site, Stack Overflow.

Lastly, we include a future work discussion in Chapter 6 to demonstrate how to extract file format constraints from documentation automatically to improve a symbolic execution tool for bug detection.

References

[1] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. Bug characteristics in open source software. *Empirical Software Engineering*, 19(6):1665–1705, December 2014.

[2] Sergio Cozzetti B. de Souza, Nicolas Anquetil, and Káthia M. de Oliveira. A study of the documentation essential to software maintenance. In *International Conference on Design of Communication*, pages 68–75, 2005.

[3] K. K. Aggarwal, Y. Singh, and J. K. Chhabra. An integrated measure of software maintainability. In *Reliability and Maintainability Symposium*, pages 235–241, 2002.

[4] L. Prechelt, B. Unger-Lamprecht, M. Philippsen, and W. F. Tichy. Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. *IEEE Transactions on Software Engineering*, 28(6):595–606, 2002.

[5] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, et al. *Fundamental concepts of dependability*. University of Newcastle upon Tyne, Computing Science, 2001.

[6] GitHub. https://github.com/, 2018.

[7] E. Wong, Taiyue Liu, and Lin Tan. Clocom: Mining existing source code for automatic comment generation. In *Software Analysis, Evolution, and Reengineering*, pages 380–389, 2015.

[8] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /*icomment: Bugs or bad comments?*/. In *ACM SIGOPS Symposium on Operating Systems Principles*, pages 145–158, 2007.

[9] R. Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, S. Oney, and A. Paradkar. Inferring method specifications from natural language api descriptions. In *International Conference on Software Engineering*, pages 815–825, 2012.

[10] Mira Kajko-Mattsson. A survey of documentation practice within corrective maintenance. *Empirical Software Engineering*, 10(1):31–55, 2005.

[11] Hao Zhong and Zhendong Su. Detecting api documentation errors. In *Object Oriented Programming Systems Languages & Applications*, pages 803–816, 2013.

[12] Lin Shi, Hao Zhong, Tao Xie, and Mingshu Li. An empirical study on evolution of api documentation. In *Fundamental Approaches to Software Engineering*, pages 416–431, 2011.

[13] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. *Programming Language Design and Implementation*, 43(6):206–215, 2008.

[14] Rupak Majumdar and Ru-Gang Xu. Directed test generation using symbolic grammars. In *Automated Software Engineering*, pages 134–143, 2007.

[15] Chromium. 134551 - pdf: Null ptr crash trying to open (corrupt?) document with two trailers. https://bugs.chromium.org/p/chromium/issues/detail?id=134551, 2012.

[16] National Vulnerability Database. Cve-2018-3924 detail. https://nvd.nist.gov/vuln/detail/CVE-2018-3924, 2018.

[17] National Vulnerability Database. Cve-2018-3939 detail. https://nvd.nist.gov/vuln/detail/CVE-2018-3939, 2018.

[18] National Vulnerability Database. Cve-2017-14458 detail. https://nvd.nist.gov/vuln/detail/CVE-2017-14458, 2017.

[19] Zero Day Initiative. Foxit reader combobox format event use-after-free remote code execution vulnerability. https://www.zerodayinitiative.com/advisories/ZDI-18-694/, 2018.

[20] Zero Day Initiative. Foxit reader resetform use-after-free remote code execution vulnerability. https://www.zerodayinitiative.com/advisories/ZDI-18-695/, 2018.

[21] Haryadi S. Gunawi, Vijayan Prabhakaran, Swetha Krishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving file system reliability with i/o shepherding. In *ACM SIGOPS Symposium on Operating Systems Principles*, pages 293–306, 2007.

[22] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. *Transactions on Computer Systems*, 24(4):393–423, 2006.

[23] Mozilla. Bug 560346 - pdf attachments corrupt when downloaded through imap. https://bugzilla.mozilla.org/show_bug.cgi?id=560346, 2016.

[24] US-CERT/NIST. Vulnerability summary for cve-2009-3608. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-3608, 2017.

[25] Launchpad. Bug #713325 "save as pdf produces broken pdf with blurred paths". https://bugs.launchpad.net/inkscape/+bug/713325, 2011.

[26] Tomasz Kuchta, Cristian Cadar, Miguel Castro, and Manuel Costa. Docovery: Toward generic automatic document recovery. In *Automated Software Engineering*, pages 563–574, 9 2014.

[27] Karl Wüst, Petar Tsankov, Saša Radomirović, and Mohammad Torabi Dashti. Force open: Lightweight black box file repair. In *DFRWS Europe*, pages S75–S82, 2017.

[28] Sumit Narayan, John A. Chandy, Samuel Lang, Philip Carns, and Robert Ross. Uncovering errors: The cost of detecting silent data corruption. In *Workshop on Petascale Data Storage*, pages 37–41, 2009.

[29] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An analysis of latent sector errors in disk drives. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 289–300, 2007.

[30] Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Garth R. Goodson, and Bianca Schroeder. An analysis of data corruption in the storage stack. *Trans. Storage*, 4(3):8:1–8:28, November 2008.

[31] Joel Spolsky and Jeff Atwood. StackOverflow. http://stackoverflow.com, 2018.

[32] E. Wong, Jinqiu Yang, and Lin Tan. Autocomment: Mining question and answer sites for automatic comment generation. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 562–567, Nov 2013.

[33] Paul W. McBurney and Collin McMillan. Automatic documentation generation via source code summarization of method context. In *International Conference on Program Comprehension*, pages 279–290, 2014.

[34] Kristina Toutanova, Dan Klein, Christopher D. Manning, and Yoram Singer. Feature-rich part-of-speech tagging with a cyclic dependency network. In *North American Chapter of the Association for Computational Linguistics on Human Language Technology - Volume 1*, pages 173–180, 2003.

[35] Dan Klein and Christopher D. Manning. Accurate unlexicalized parsing. In *Proceedings of Annual Meeting on Association for Computational Linguistics - Volume 1*, pages 423–430, 2003.

[36] Manziba Akanda Nishi and Kostadin Damevski. Scalable code clone detection and search based on adaptive prefix filtering. *Journal of Systems and Software*, 137:130 – 142, 2018.

[37] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. Sourcerercc: Scaling code clone detection to big-code. In *International Conference on Software Engineering*, pages 1157–1168, 2016.

[38] L.A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, SE-2(3):215–222, 1976.

[39] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[40] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX conference on Operating systems design and implementation*, pages 209–224, 2008.

[41] P. D. Marinescu and C. Cadar. KATCH: High-coverage testing of software patches. In *Joint Meeting on Foundations of Software Engineering*, pages 235–245, 2013.

[42] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *Architectural Support for Programming Languages and Operating Systems*, pages 265–278, 2011.

[43] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.

[44] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta. Symbolic pathfinder: Integrating symbolic execution with model checking for java bytecode analysis. *Automated Software Engineering*, 20(3):391–425, 2013.

[45] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 513–528, 2010.

[46] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Foundations of Software Engineering*, pages 263–272, 2005.

[47] Nikolai Tillmann and Jonathan De Halleux. Pex–white box test generation for. net. In *Tests and Proofs*, pages 134–153. 2008.

[48] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen. Directed test suite augmentation: Techniques and tradeoffs. In *Foundations of Software Engineering*, pages 257–266, 2010.

[49] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S Foster, and Michael Hicks. Directed symbolic execution. In *International Conference on Static Analysis*, pages 95–111. 2011.

[50] David M. Perry, Andrea Mattavelli, Xiangyu Zhang, and Cristian Cadar. Accelerating array constraints in symbolic execution. In *International Symposium on Software Testing and Analysis*, pages 68–78, 2017.

[51] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. Multise: Multi-path symbolic execution using value summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 842–853, New York, NY, USA, 2015. ACM.

[52] D. Babić, L. Martignoni, S. McCamant, and D. Song. Statically-directed dynamic automated test generation. In *International Symposium on Software Testing and Analysis*, pages 12–22, 2011.

[53] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing symbolic execution with veritesting. In *International Conference on Software Engineering*, pages 1083–1094, 2014.

[54] Y. Li, Z. Su, L. Wang, and X. Li. Steering symbolic execution to less traveled paths. In *International Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 19–32, 2013.

[55] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *International Conference on Automated Software Engineering*, pages 443–446, 2008.

[56] C. Y. Cho, D. Babić, P. Poosankam, K. Z. Chen, E. X. Wu, and D. Song. MACE: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *USENIX conference on Security*, pages 10–10, 2011.

[57] K. Krishnamoorthy, M. S. Hsiao, and L. Lingappan. Strategies for scalable symbolic execution-driven test generation for programs. *Science China Information Sciences*, 54(9):1797–1812, 2011.

[58] R. Santelices and M. J. Harrold. Exploiting program dependencies for scalable multiple-path symbolic execution. In *International Symposium on Software Testing and Analysis*, pages 195–206, 2010.

[59] Sangmin Park, B. M. Mainul Hossain, Ishtiaque Hussain, Christoph Csallner, Mark Grechanik, Kunal Taneja, Chen Fu, and Qing Xie. Carfast: Achieving higher statement coverage faster. In *International Symposium on the Foundations of Software Engineering*, pages 35:1–35:11, 2012.

[60] P. D. Marinescu and C. Cadar. Make test-zesti: A symbolic execution solution for improving regression testing. In *International Conference on Software Engineering*, pages 716–726, 2012.

[61] E. J. Weyuker and T. J. Ostrand. Theories of program testing and the application of revealing subdomains. *Transactions on Software Engineering*, SE-6(3):236–246, 1980.

[62] R. Majumdar and R. Xu. Reducing test inputs using information partitions. In *International Conference on Computer Aided Verification*, pages 555–569, 2009.

[63] M. Staats and C. Păsăreanu. Parallel symbolic execution for structural test generation. In *International symposium on Software testing and analysis*, pages 183–194, 2010.

[64] D. Qi, H. D.T. Nguyen, and A. Roychoudhury. Path exploration based on symbolic output. In *ACM Transactions on Software Engineering and Methodology*, pages 278–288, 2011.

[65] David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. Chopped symbolic execution. In *International Conference on Software Engineering*, pages 350–360, 2018.

[66] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *USENIX Security Symposium*, 2018.

[67] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Network and Distributed System Security Symposium*, 2016.

[68] Carlos Pacheco. *Directed Random Testing*. Ph.D., MIT Department of Electrical Engineering and Computer Science, Cambridge, Massachusetts, 2009.

[69] Brian Demsky and Martin Rinard. Automatic detection and repair of errors in data structures. In *ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, pages 78–95, 2003.

[70] G. Endignoux, O. Levillain, and J. Y. Migeon. Caradoc: A pragmatic approach to pdf parsing and validation. In *IEEE Security and Privacy Workshops*, pages 126–139, 2016.

[71] Fan Long, Vijay Ganesh, Michael Carbin, Stelios Sidiroglou, and Martin Rinard. Automatic input rectification. In *International Conference on Software Engineering*, pages 80–90, 2012.

[72] Martin C. Rinard. Living in the comfort zone. In *ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, pages 611–622, 2007.

[73] Artifex Software. Mupdf. http://mupdf.com/news, 2016.

[74] PDF Labs. Pdftk - the pdf toolkit. https://www.pdflabs.com/tools/pdftk-the-pdf-toolkit/, 2016.

[75] SysInfoTools Software. Pdf recovery tool. https://gallery.technet.microsoft.com/PDF-Repair-Tool-to-Restore-f21703c9.

[76] Inc Repair Toolbox. Pdf repair toolbox. https://gallery.technet.microsoft.com/PDF-Repair-Tool-to-Restore-f21703c9.

[77] Freedesktop. Poppler. https://poppler.freedesktop.org/, 2017.

[78] GhostScript. Bugzilla – main page. http://bugs.ghostscript.com/, 2016.

[79] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. In *Foundations of Software Engineering*, pages 263–272, 2005.

[80] Brian Demsky and Martin Rinard. Automatic detection and repair of errors in data structures. *ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, 38(11):78–95, 2003.

[81] Ralf D. Brown. Reconstructing corrupt deflated files. *Digital Investigation: The International Journal of Digital Forensics & Incident Response*, 8:S125–S131, 2011.

[82] Ralf D. Brown. Improved recovery and reconstruction of deflated files. *DFRWS Conference*, 10:S21–S29, 2013.

[83] K.K. Aggarwal, Y. Singh, and J.K. Chhabra. An integrated measure of software maintainability. In *Proceedings of Annual Reliability and Maintainability Symposium*, pages 235–241, 2002.

[84] S. Zhang, C. Zhang, and M. D. Ernst. Automated documentation inference to explain failed tests. In *Automated Software Engineering*, pages 63–72, 2011.

[85] Raymond P. L. Buse and Westley R. Weimer. Automatic documentation inference for exceptions. In *International Symposium on Software Testing and Analysis*, pages 273–282, 2008.

[86] F. Long, X. Wang, and Y. Cai. Api hyperlinking via structural overlap. In *Foundations of Software Engineering*, pages 203–212, 2009.

[87] R. P. L. Buse and W. Weimer. Automatically documenting program changes. In *Automated Software Engineering*, pages 33–42, 2010.

[88] G. Sridhara, L. Pollock, and K. Vijay-Shanker. Generating parameter comments and integrating with method summaries. In *International Conference on Program Comprehension*, pages 71–80, 2011.

[89] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Automated Software Engineering*, pages 43–52, 2010.

[90] G. Sridhara, L. Pollock, and K. Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *International Conference on Software Engineering*, pages 101–110, 2011.

[91] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for java classes. In *International Conference on Program Comprehension*, pages 23–32, 2013.

[92] E. Hill, L. Pollock, and K. Vijay-Shanker. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *International Conference on Software Engineering*, pages 232–242, 2009.

[93] S. Rastkar, G. C. Murphy, and A. W. J. Bradley. Generating natural language summaries for crosscutting source code concerns. In *International Conference on Software Maintenance*, pages 103–112, 2011.

[94] Suparna Gundagathi Manjunath. Towards comment generation for mpi programs. Master's thesis, University of Delaware, 2011.

[95] X. Wang, L. Pollock, and K. Vijay-Shanker. Automatic segmentation of method code into meaningful blocks to improve readability. In *Working Conference on Reverse Engineering*, pages 35 –44, 2011.

[96] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *International Conference on Program Comprehension*, ICPC '18, pages 200–210, 2018.

[97] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke S. Zettlemoyer. Summarizing source code using a neural attention model. In *ACL*, 2016.

[98] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus. On the use of automated text summarization techniques for summarizing source code. In *Working Conference on Reverse Engineering*, pages 35–44, 2010.

[99] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella. Using ir methods for labeling source code artifacts: Is it worthwhile? In *International Conference on Program Comprehension*, pages 193–202, 2012.

[100] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. Supporting program comprehension with source code summarization. In *International Conference on Software Engineering*, pages 223–226, 2010.

[101] B. P. Eddy, J. A. Robinson, N. A. Kraft, and J. C. Carver. Evaluating source code summarization techniques: Replication and expansion. In *International Conference on Program Comprehension*, pages 13–22, 2013.

[102] Paul W. McBurney, Cheng Liu, Collin McMillan, and Tim Weninger. Improving topic model source code summarization. In *International Conference on Program Comprehension*, pages 291–294, 2014.

[103] Laura Moreno and Jairo Aponte. On the analysis of human and automatic summaries of source code. *CLEI Electronic Journal*, 15(2), 2012.

[104] S. Panichella, J. Aponte, M. D. Penta, A. Marcus, and G. Canfora. Mining source code descriptions from developer communications. In *International Conference on Program Comprehension*, pages 63–72, 2012.

[105] B. Dagenais and M. P. Robillard. Recovering traceability links between an api and its learning resources. In *International Conference on Software Engineering*, pages 47–57, 2012.

[106] Carmine Vassallo, Sebastiano Panichella, Massimiliano Di Penta, and Gerardo Canfora. Codes: Mining source code descriptions from developers discussions. In *International Conference on Program Comprehension*, pages 106–109, 2014.

[107] J. Kim, S. Lee, S. Hwang, and S. Kim. Enriching documents with examples: A corpus mining approach. *ACM Transactions on Information Systems*, pages 1:1–1:27, 2013.

[108] A. Bacchelli, M. D'Ambros, and M. Lanza. Extracting source code from e-mails. In *International Conference on Program Comprehension*, pages 24 –33, 2010.

[109] Nicolas Bettenburg, Bram Adams, Ahmed E. Hassan, and Michel Smidt. A lightweight approach to uncover technical artifacts in unstructured data. In *International Conference on Program Comprehension*, pages 185–188, 2011.

[110] P. Chatterjee, M. A. Nishi, K. Damevski, V. Augustine, L. Pollock, and N. A. Kraft. What information about code snippets is available in different software-related documents? an exploratory study. In *International Conference on Software Analysis, Evolution and Reengineering*, pages 382–386, 2017.

[111] P. Chatterjee, B. Gause, H. Hedinger, and L. Pollock. Extracting code segments and their descriptions from research articles. In *International Conference on Mining Software Repositories*, pages 91–101, 2017.

[112] Dick Grune. The software and text similarity tester SIM. http://dickgrune.com/Programs/similarity_tester/, 2015.

[113] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, pages 176 – 192, 2006.

[114] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, pages 654–670, 2002.

[115] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *International Conference on Software Engineering*, pages 96–105, 2007.

[116] I.D. Baxter, C. Pidgeon, and M. Mehlich. Dms®: Program transformations for practical scalable software evolution. In *International Conference on Software Engineering*, pages 625–634, 2004.

[117] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *International Symposium on Static Analysis*, pages 40–56, 2001.

[118] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, pages 470–495, 2009.

[119] R. Wettel and R. Marinescu. Archeology of code duplication: recovering duplication chains from small duplication fragments. In *International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 8 pp.–, 2005.

[120] Pengcheng Wang, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu, and Chanchal K. Roy. Ccaligner: A token based large-gap clone detector. In *International Conference on Software Engineering*, pages 1066–1077, 2018.

[121] I. Keivanloo, C.K. Roy, J. Rilling, and P. Charland. Shuffling and randomization for scalable source code clone detection. In *International Workshop on Software Clones*, pages 82–83, 2012.

[122] C.K. Roy and J.R. Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *International Conference on Program Comprehension*, pages 172–181, 2008.

[123] N. Gode and R. Koschke. Incremental clone detection. In *European Conference on Software Maintenance and Reengineering*, pages 219–228, 2009.

[124] Patrice Godefroid, Michael Y. Levin, and David Molnar. Sage: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27, January 2012.

[125] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.

[126] R. Majumda and R. Xu. Directed test generation using symbolic grammars. In *International Conference on Automated Software Engineering*, pages 134–143, 2007.

[127] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 322–335, New York, NY, USA, 2006. ACM.

[128] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. Perffuzz: Automatically generating pathological inputs. In *International Symposium on Software Testing and Analysis*, pages 254–265, 2018.

[129] C. Rubio-González and B. Liblit. Expect the unexpected: Error code mismatches between documentation and the real world. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 73–80, 2010.

[130] L. Tan, Y. Zhou, and Y. Padioleau. aComment: Mining annotations from comments and code to detect interrupt-related concurrency bugs. In *International Conference on Software Engineering*, pages 11–20, 2011.

[131] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens. @tComment: Testing javadoc comments to detect comment-code inconsistencies. In *International Conference on Software Testing, Verification and Validation*, pages 260–269, 2012.

[132] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring specifications for resources from natural language api documentation. *Automated Software Engineering Journal*, 18(3-4):227–261, 2011.

[133] Edmund Wong, Lei Zhang, Song Wang, Taiyue Liu, and Lin Tan. Dase: Document-assisted symbolic execution for improving automated software testing. In *International Conference on Software Engineering - Volume 1*, pages 620–631, 2015.

[134] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. Inferring resource specifications from natural language api documentation. In *Automated Software Engineering*, pages 307–318, 2009.

[135] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. Inferring method specifications from natural language api descriptions. In *International Conference on Software Engineering*, pages 815–825, 2012.

[136] Cindy Rubio-González and Ben Liblit. Expect the unexpected: Error code mismatches between documentation and the real world. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 73–80, 2010.

[137] L. C. Briand and A. Wolf. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering*, pages 85–103, 2007.

[138] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on java predicates. *SIGSOFT Software Engineering Notes*, 27(4):123–133, 2002.

[139] S. Thummalapenta, K. V. Lakshmi, S. Sinha, N. Sinha, and S. Chandra. Guided test generation for web applications. In *International Conference on Software Engineering*, pages 162–171, 2013.

[140] A. Avancini and M. Ceccato. Comparison and integration of genetic algorithms and dynamic symbolic execution for security testing of cross-site scripting vulnerabilities. *Information and Software Technology*, 55(12):2209–2222, 2013.

[141] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: Preliminary assessment. In *International Conference on Software Engineering*, pages 1066–1071, 2011.

[142] M. K. Ganai, N. Arora, C. Wang, A. Gupta, and G. Balakrishnan. BEST: A symbolic testing tool for predicting multi-threaded program failures. In *International Conference on Automated Software Engineering*, pages 596–599, 2011.

[143] K. Li, C. Reichenbach, Y. Smaragdakis, Y. Diao, and C. Csallner. SEDGE: Symbolic example data generation for dataflow programs. In *International Conference on Automated Software Engineering*, pages 235–245, 2013.

[144] P. Zhang, S. Elbaum, and M. B. Dwyer. Automatic generation of load tests. In *Automated Software Engineering*, pages 43–52, 2011.

[145] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 206–215, 2008.

[146] David McClosky and Christopher D. Manning. Learning constraints for consistent timeline extraction. In *Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 873–882, 2012.

[147] Katsumasa Yoshikawa, Sebastian Riedel, Masayuki Asahara, and Yuji Matsumoto. Jointly identifying temporal relations with markov logic. In *Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 1 - Volume 1*, pages 405–413, 2009.

[148] Achim D Brucker, Matthias P Krieger, Delphine Longuet, and Burkhart Wolff. A specification-based test case generation method for uml/ocl. In *Models in Software Engineering*, pages 334–348. 2011.

[149] J. Offutt and A. Abdurazik. Generating tests from uml specifications. In *International Conference on The Unified Modeling Language: Beyond the Standard*, pages 416–429, 1999.

[150] Sourceware Bugzilla. The Stanford natural language processing dependencies. http://nlp.stanford.edu/software/stanford-dependencies.shtml, 2015.

[151] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language api documentation. In *Automated Software Engineering*, pages 307–318, 2009.

[152] Marie-Catherine de Marneffe and Christopher D. Manning. Stanford typed dependencies manual. http://nlp.stanford.edu/software/dependencies_manual.pdf, 2013.

[153] Tool Interface Standards. Executable and linkable format. http://www.skyfree.org/linux/references/ELF_Format.pdf, 2015.

[154] The Clang Team. Clang 8 documentation. https://clang.llvm.org/docs/ClangTools.html, 2018.

[155] The KLEE Team. OSDI'08 Coreutils Experiments. http://klee.github.io/docs/coreutils-experiments/, 2015.

[156] Sourceware Bugzilla. Readelf bug 16664 - Segmentation fault in process_attributes() of readelf.c. https://sourceware.org/bugzilla/show_bug.cgi?id=16664, 2014.

[157] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007.

[158] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *International Conference on Software Engineering*, pages 291–301, 2002.

[159] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *ACM Symposium on Operating Systems Principles*, pages 57–72, 2001.

[160] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *International Conference on Software Engineering*, pages 474–484, 2009.

[161] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling black-box mutational fuzzing. In *ACM Conference on Computer and Communications Security*, pages 511–522, 2013.

[162] American fuzzy lop. http://lcamtuf.coredump.cx/afl/, 2018.

[163] Glyph & Cog. pdftotext. https://www.xpdfreader.com/pdftotext-man.html, 2018.

[164] F. Zhao. On choosing the digital document's file format for long-term preservation. In *International Conference on Communication Software and Networks*, pages 370–372, 2011.

[165] M. Y. B. Masod and R. Ahmad. Digital data exchange in digital prepress technology: The adoption of a new standards. In *IEEE Business Engineering and Industrial Applications Colloquium*, pages 410–414, 2013.

[166] Adobe Systems Incorporated. Pdf reference and adobe extensions to the pdf specification, 2016.

[167] Adobe Systems Incorporated. Document management - portable document format - part 1: Pdf 1.7. http://www.adobe.com/devnet/pdf/pdf_reference.html, 2017.

[168] Tomasz Kuchta, Thibaud Lutellier, Edmund Wong, Lin Tan, and Cristian Cadar. On the correctness of electronic documents: studying, finding, and localizing inconsistency bugs in pdf readers and files. *Empirical Software Engineering*, 2018.

[169] Scrapinghub. Scrapy. http://scrapy.org/, 2016.

[170] Simson Garfinkel, Paul Farrell, Vassil Roussev, and George Dinolt. Bringing science to digital forensics with standardized forensic corpora. In *The Digital Forensic Research Conference*, 2009.

[171] GNOME. Documents - document manager. https://wiki.gnome.org/Apps/Documents, 2018.

[172] Okular developers. Okukar. https://okular.kde.org/, 2016.

[173] Apache. The apache fop project. https://xmlgraphics.apache.org/fop/, 2018.

[174] LibreOffice. Bugzilla main page. https://bugs.documentfoundation.org/, 2016.

[175] Lincoln. Pdftohtml conversion program. http://pdftohtml.sourceforge.net/, 2018.

[176] Mozilla Foundation. Bugzilla. https://www.bugzilla.org/, 2016.

[177] Google. Monorail issue tracker. https://chromium.googlesource.com/infra/infra/+/master/appengine/monorail, 2016.

[178] Canonical Group Ltd. Launchpad. https://launchpad.net/, 2016.

[179] Adobe. Adobe acrobat pro dc. https://acrobat.adobe.com/us/en/acrobat/acrobat-pro.html, 2018.

[180] Brendan Zagaeski. Minimal pdf - contents. http://brendanzagaeski.appspot.com/0004.html, 2018.

[181] Artifex Software. Ghostscript. http://www.ghostscript.com/, 2016.

[182] The GNOME Project. Apps/evince. https://wiki.gnome.org/Apps/Evince, 2016.

[183] Glyph & Cog. Xpdf: A pdf viewer for x. https://wiki.gnome.org/Apps/Evince, 2016.

[184] Google. The chromium projects. https://www.chromium.org/Home, 2016.

[185] Google. Pdfium. https://pdfium.googlesource.com/pdfium/, 2017.

[186] Mozilla. Firefox. http://mupdf.com/news, 2016.

[187] Mozilla. Pdf.js. https://mozilla.github.io/pdf.js/, 2017.

[188] Adobe Systems Incorporated. Adobe acrobat reader dc. https://get.adobe.com/reader/, 2016.

[189] Adam Reichold. qpdfview. https://launchpad.net/qpdfview, 2016.

[190] Volker C. Behr. Cups-pdf. https://www.cups-pdf.de/, 2015.

[191] Chromium. Bug # "out of bounds write in pdf with sample function with lots of inputs". https://bugs.chromium.org/p/chromium/issues/detail?id=124182, 2012.

[192] Gavriel Salvendy. Book review understanding your users: A practical guide to user requirements by catherine courage and kathy baxter. *International Journal of Human–Computer Interaction*, 19(1):155–156, 2005.

[193] Enough Pepper. xsort - free card sorting application for mac. https://xsortapp.com/, 2018.

[194] GNOME. Bug #349826 "gtk+ produces invalid pdf on amd64". https://bugzilla.gnome.org/show_bug.cgi?id=349826, 2006.

[195] Document Foundation. Bug 39946 - different appearance while printing or exporting to pdf. https://bugs.documentfoundation.org/show_bug.cgi?id=39946, 2011.

[196] Freedesktop. Bug 43646 - downloaded pdf w/forms will not open when uploaded to adobe. https://bugs.freedesktop.org/show_bug.cgi?id=43646, 2011.

[197] Launchpad. Bug 518230 - some pdf forms don't save entered information. https://bugs.launchpad.net/ubuntu/+source/evince/+bug/518230, 2010.

[198] Chromium. Bug 179013 - print blank page on chrome. https://bugs.chromium.org/p/chromium/issues/detail?id=179013, 2013.

[199] Launchpad. Bug #423630 "lists the right application but failt to run it when mimetype wrongly set on server". https://bugs.launchpad.net/ubuntu/+source/firefox/+bug/423630, 2010.

[200] Launchpad. Simple scan. https://launchpad.net/simple-scan, 2018.

[201] Launchpad. Bug 760967 - simple scan creates corrupt file and freezes when saving pdf. https://bugs.launchpad.net/ubuntu/+source/simple-scan/+bug/760967, 2011.

[202] Launchpad. Bug 789906 - crash when saving pdf. https://bugs.launchpad.net/ubuntu/+source/simple-scan/+bug/789906, 2011.

[203] Launchpad. Bug #537331 "evince crashed with sigsegv in __memset_sse2() when opening a pdf". https://bugs.launchpad.net/ubuntu/+source/poppler/+bug/537331, 2010.

[204] ICC. International color consortium. http://www.color.org/index.xalter, 2018.

[205] Chromium. saving pdfs results in damaged file. https://bugs.chromium.org/p/chromium/issues/detail?id=70440, 2011.

[206] Beat Fluri, Michael Wursch, and Harald C. Gall. Do code and comments co-evolve? on the relation between source code and comment changes. In *Working Conference on Reverse Engineering*, pages 70–79, 2007.

[207] Paul W. McBurney and Collin McMillan. Automatic documentation generation via source code summarization of method context. In *International Conference on Program Comprehension*, pages 279–290, 2014.

[208] E. Wong, Jinqiu Yang, and Lin Tan. Autocomment: Mining question and answer sites for automatic comment generation. In *Automated Software Engineering*, pages 562–567, 2013.

[209] Chris Parnin, Christoph Treude, Lars Grammel, and Margaret-Anne D. Storey. Crowd documentation: Exploring the coverage and the dynamics of api discussions on stack overflow. Technical Report GIT-CS-12-05, Georgia Tech, 2012.

[210] S. Subramanian and R. Holmes. Making sense of online code snippets. In *Mining Software Repositories*, pages 85–88, 2013.

[211] Edmund Wong, Jinqiu Yang, Nasir Ali, and Lin Tan. Autocomment: Crowd sourced comment generation. http://asset.uwaterloo.ca/AutoComment2/, 2018.

[212] Inc. Stack Exchange. Stack exchange data dump. https://archive.org/details/stackexchange, 2018.

[213] L. Ponzanelli, A. Mocci, and M. Lanza. Stormed: Stack overflow ready made data. In *Working Conference on Mining Software Repositories*, pages 474–477, 2015.

[214] L. Ponzanelli, A. Mocci, A. Bacchelli, M. Lanza, and D. Fullerton. Improving low quality stack overflow post detection. In *International Conference on Software Maintenance and Evolution*, pages 541–544, 2014.

[215] Beatrice Santorini. Part-of-speech tagging guidelines for the penn treebank project (3rd revision, 2nd printing). Technical report, Department of Linguistics, University of Pennsylvania, 1990.

[216] Di Yang, Aftab Hussain, and Cristina Videira Lopes. From query to usable code: An analysis of stack overflow code snippets. In *Mining Software Repositories*, pages 391–402, 2016.

[217] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. On detection of gapped code clones using gap locations. In *Software Engineering Conference*, pages 327–336, 2002.

[218] B. Li, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and N. A. Kraft. Automatically documenting unit test cases. In *International Conference on Software Testing, Verification and Validation*, pages 341–352, 2016.

[219] Al Danial. CLOC – count lines of code. https://github.com/AlDanial/cloc, 2016.

[220] B. D. Cruz, P. W. McBurney, and C. McMillan. Tracelab components for reproducing source code summarization experiments. In *International Conference on Software Maintenance and Evolution*, pages 610–610, 2016.

[221] R. Likert. A technique for the measurement of attitudes. *Archives of Psychology*, 22(140):1–55, 1932.

[222] Luis Fernando Cortés-Coy, Mario Linares-Vásquez, Jairo Aponte, and Denys Poshyvanyk. On automatically generating commit messages via summarization of source code changes. In *International Working Conference on Source Code Analysis and Manipulation*, pages 275–284, 2014.

[223] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrian Marcus, and Gerardo Canfora. Automatic generation of release notes. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 484–495, New York, NY, USA, 2014. ACM.

[224] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: A method for automatic evaluation of machine translation. In *Association for Computational Linguistics*, pages 311–318, 2002.

[225] Siyuan Jiang, Ameer Armaly, and Collin McMillan. Automatically generating commit messages from diffs using neural machine translation. *CoRR*, 2017.

[226] J.L. Fleiss et al. Measuring nominal scale agreement among many raters. *Psychological Bulletin*, 76(5):378–382, 1971.

[227] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. The java language specification java se 7 edition. https://docs.oracle.com/javase/specs/jls/se7/jls7.pdf, 2013.

[228] Jue Wang, Yingnong Dang, Hongyu Zhang, Kai Chen, Tao Xie, and Dongmei Zhang. Mining succinct and high-coverage api usage patterns from source code. In *Mining Software Repositories*, 2013.

[229] Dave Binkley, Dawn Lawrie, Emily Hill, Janet Burge, Ian Harris, Regina Hebig, Oliver Keszocze, Karl Reed, and John Slankas. Task-driven software summarization. In *International Conference on Software Maintenance*, pages 432–435, 2013.

[230] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. Hampi: A solver for string constraints. In *International Symposium on Software Testing and Analysis*, pages 105–116, 2009.