# INTENSIFICATION OF SOFTWARE SECURITY USING SECURE OBFUSCATION WITH INJECTING AND DETECTING CODE CLONES

*A THESIS SUBMITTED IN FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF*

## DOCTOR OF PHILOSOPHY

By

**PRATIKSHA GAUTAM**


**Department of Computer Science & Engineering and Information Technology**

**JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY, WAKNAGHAT, SOLAN-173234, HIMACHAL PRADESH, INDIA**

**MAY, 2019**

# TABLE OF CONTENTS

# DECLARATION BY THE SCHOLAR

I hereby declare that the work reported in the Ph.D. thesis entitled **"Intensification of Software Security using Secure Obfuscation with Injecting and Detecting Code Clones"** submitted to the Department of Computer Science Engineering and Information Technology, **Jaypee University of Information Technology (JUIT), Waknaghat, India,** is an authentic record of my work carried out under the supervision of **Dr. Hemraj Saini,** Associate Professor, JUIT. The work in this thesis is my original investigation and has not been submitted elsewhere for the award of any other degree or diploma. I am fully responsible for the contents of my Ph.D. Thesis.

Pratiksha Gautam
Department of Computer Science and Engineering
Jaypee University of Information Technology,
Waknaghat, India
Date:

# SUPERVISOR'S CERTIFICATE

This is to certify that the work reported in the Ph.D. thesis entitled **"Intensification of Software Security using Secure Obfuscation with Injecting and Detecting Code Clones"** submitted by **Pratiksha Gautam** at **Jaypee University of Information Technology, Waknaghat**, **India,** is a bonafide record of her original work carried out under my supervision. This work has not been submitted elsewhere for any other degree or diploma.

Dr. Hemraj Saini
Department of Computer Science and Engineering
Jaypee University of Information Technology,
Waknaghat, India
Dated:

# ACKNOWLEDGEMENTS

First of all, I would like to express my heart-felt and most sincere gratitude to my respected supervisor, Dr. Hemraj Saini for his constant guidance, advice, encouragement and extraordinary patience during this thesis work. Without him, this work would have been impossible, and he has in fact further unlocked the research potential within me.

I would like to devote my very special thanks particularly to Dr. Sanjiv Kumar Tiwari in JUIT who guided and supported me during copious important phases and decisions. I also thank Dr. Dhavleesh Rattan for his help and suggestions during the thesis work.

I would like to thank Prof. Dr. Samir Dev Gupta, Director & Academic Head and Prof. Dr. Vinod Kumar, Vice Chancellor, Jaypee University of Information Technology, Waknaghat, for providing admirable research atmosphere and amenities for my research. Thanks are also due to Prof. Dr. Satya Prakash Ghrera, Head of Computer Science & Engineering and Information & Communication Technology, Waknaghat, for his helpful comments, insights and suggestions.

I am grateful to Dr. Swarup Roy, Sikkim University for his overwhelming support that helped me to concentrate more deeply on my thesis work.

I thank the anonymous reviewers for their valuable comments and suggestions in improving the papers produced from this thesis. I also thank the tool authors who provided useful answers to our queries.

I would like to thank all of my friends who have helped me in one way or another along the way.

I am ceaselessly appreciative my gratefulness to my family members and relatives especially, my affectionate younger brothers Dr. Gaurav Gautam and Mr. Dushyant Gautam for their support and inspiration.

The most wonderful thing of my life is my father Mr. Ashok Kumar Gautam, and my mother Mrs. Sudesh Gautam who drove me in the right direction in successfully finishing my thesis.

Thanks to God for their love for us.

Pratiksha Gautam

Department of Computer Science and Engineering

Jaypee University of Information Technology,

Waknaghat, India

Date:

# ABSTRACT

Software security plays a vital role in Information Technology as it's infringe lead to tremendous monetary depletions due to the resale of application thus; it is a potential research issue in the present scenario. It can be classified as coding errors and copyright protection of software. There are three types of copyright protection attacks as software piracy, reverse engineering, and tampering. The security approaches against these assaults are software watermarking, code obfuscation and tamper-proofing. The code obfuscation is one of the pioneer methods of software protection against reverse engineering; obfuscate a code into a form which is thornier for an attacker to comprehend or modify the original code from the illegal reverse engineering procedure. The code transformation assaults can be categorized as 1) static analysis, 2) dynamic analysis, and 3), code clone attack. The reverse engineering attacks occur due to the unconfined software code to the user. Malicious reverse engineering of software codes can be compact by the travail of code obfuscation on source code. None of the existing code transformation technique protects from reverse engineering assaults. A novel code transformation method is needed for software security. Therefore, to avert static analysis assault and dynamic analysis attack on obfuscation and to make code compact for an adversary. Farther, the code clone (non-trivial software clone) is used to thwart static and dynamic analysis attacks on obfuscation. The code clone detection is another emerging research issue in software clone detection area because of it is considered adverse in software evolution and maintenance. Though various software clone detection tools and techniques have been proposed in the literature, however, in spite vigorous study there is conspicuous inadequacy of exertion in the identification and study of type-1 to type-4 software clones. Although, software clones can be classified by their features as textually equivalent software clone or functionally identical code clones. Furthermore, syntactic equivalent clones are categorized into type-1 to type-3 while semantically equivalent code clones are type-4. The type-1 software clones are identical code fragments excluding few amendments in whitespaces, blank spaces, comments, etc. while type-2 is similar to original segments except renaming identifiers. Type-3 clones are those where to insert or delete new lines in the source code, and type-4 software clones have similar functionality with the different structure. Thus, in this thesis firstly a state-of-art in software security as well as in software clones detection and their analysis in various manners. Foremost, to proposed hybrid approach which is based on tokenization concept and it can detect type-1(similar code fragments with some adoptions as whitespaces, blank space, comments, etc.)

with high precision and recall. Subsequently, proposed an approach for the detection of type-2(renaming identifiers with comments, white spaces, etc.) software clones with the help of a directed acyclic graph which is a compiler optimization technique.

Moreover, the proposed approach is considered to deal with the distinct types of software clones using the proposed editing mutation operator-based taxonomy. Third, we develop a method for the detection of trivial- software clones by using an approach namely as a control flow graph (CFG) and reduced flow graph (RFG). The non-trivial software clones detected by using program dependency graph (PDG) . Fourth, to evaluate the performance of clone detection tools and techniques from two perspectives. Foremost, to evaluate regarding software metrics and subsequently, by generated test cases. Fifth, a mutation operators-based editing nomenclature for replica generation that replica developers' expurgation behaviors in the copy/pasted code in a top-down manner. Subsequently, by using proposed taxonomy, we accomplished a scenario-based qualitative assessment and evaluation of the existing clone tools and approaches for evaluating software clone detection tools and techniques. Furthermore, by using the result of this research, the users can choose right tool according to their requirements as well as it shows the constraints of a peculiar approach. Sixth, in order to evaluate software clone detection tools and techniques in rational manner and to evade challenge and manual massive endeavor for injecting software clones in source code as well as for validating detection tools and techniques, we have proposed an automatic mutation operator-based evaluation framework that automatically injects type-1 to type-2 code clones in the program text. Furthermore, it measures the detection tools precision and recalls with the minimum threshold value. This study has revealed that proposed approach is adequate for the detection of type-1 to type-3 software clones which are created with the help of mutation operator-based editing classification in large-scale application system with high precision and recall.

# LIST OF ABBREVIATIONS

| | |
|---|---|
| AST | Abstract Syntax Tree |
| BSA | Business Software Alliance |
| BRIC | Brazil, Russia, India, South Africa and China |
| CC | Cyclomatic Complexity |
| CFG | Control Flow Graph |
| CS | Code Segment |
| CPD | Copy Paste Detector |
| DAG | Direct Acyclic Graph |
| DDG | Data Dependency Graph |
| FDT | Forward Dominance Tree |
| H | High |
| IRL | Intermediate Representation Language |
| IPR | Intellectual Property Right |
| LOC | Line of Code |
| LSH | Locative Sensitive Hashing |
| M | Medium |
| MO | Mutation Operators |
| mACP | Arbitrary Changes in Parameters |
| mCNWs | Add New line Spaces |
| mCWs | Changes in White Spaces |
| mMCs | Alterations in Comments |
| mARDT | Arbitrary Renaming of Data Types |
| mARV | Arbitrary Renaming of Variables |
| mARL | Arbitrary Renaming of Literals |
| mAOR | Changes in Arithmetic Operators |
| mCB | Changes in Blank Spaces |
| mCC | Changes in Comments |

| | |
|---|---|
| mCF | Changes in Formatting |
| mDSV | Variation Data Statement |
| mioCB | Mutated Code Base |
| moCS | Mutated Code Segment |
| mMLs | Modification in the Whole Line |
| mSDL | Small Deletions within a Line |
| mSIL | Small Insertions within A Line |
| mROS | Reorder the Statement |
| mRPE | Replacement of Parameters with Expressions |
| mRW | Removing Whitespaces |
| mSR | Systematic Renaming of Identifiers |
| mTBs | Changes in Blank Spaces |
| mVF | Change the Formatting |
| oCB | Original Code Base |
| NICAD | Near-Miss Intentional Clones |
| PDG | Program Dependency Graph |
| PMD | Programming Mistake Detector |
| RFG | Reduced Flow Graph |
| SDD | Similar Data Detection |
| TCS | Tata Consultancy Service |
| THLV | Threshold Level Value |

# LIST OF FIGURES

# LIST OF TABLES

# CHAPTER 1

# INTRODUCTION

In this chapter, we confers this thesis a brief description. The motivation of this thesis is presented in Section 1.1. After indulging motivation, we explain the problems, research objective in section 1.2 and section 1.3, respectively. We summarize our augmentations to those research gaps detailed in Section 1.4. Finally, section 1.5 present a layout of the remaining chapters.

## 1.1 MOTIVATION

The piracy of Software, illegal reproduction, and resale of software applications is $ 62.7 billion in 2013 and more than $ 400 [1] from cyber-attacks in 2015. The current study of BSA [2] and joint research which is accomplished by National University of Singapore and IDC [3] presents that malicious code is found in the pirated software. The survey report also illustrates per year piracy rate as shown in Table 1 and that due to this pirated software we have spent millions of dollar per year for software protection which is shown in Table 2. Thus, malware is a big concern in the software industry and pirated software, or unlicensed software is one of the causes of malware. Therefore, it is necessary to secure code from software piracy. Although, various software code protection methods have been evolved for intellectual property right protection as software watermarking, obfuscation and tampering [4-5]. Code transformation is one of the security techniques of Reverse Engineering which protects source code from reverse engineering attacks by making code harder for an adversary [6-8]. The reverse engineering attacks have three types as a static attack (it comprehends the functionality of software by statically analyzing code), dynamic attack (to run software with different inputs) and code clone attack (to detect and remove the duplicate code from source code) [9]. Therefore, the first research objective focuses on securing code obfuscation from static and dynamic attacks using code clone attack. The second objective focus on code clone attacks in which duplicate codes have been inserted by copying/pasting activity in source code segments during the software development. To rewrite a code fragment with some changes are known as code replication and copied code is called code clone.

**Table 1.1:** 2016 BSA piracy study (piracy rate)

| S. No | 2015 | 2013 | 2011 | 2009 |
|---|---|---|---|---|
| India | 58% | 60% | 63% | 65% |
| Total AP | 61% | 62% | 60% | 59% |
| Central and Eastern Europe | 58% | 61% | 62% | 64% |
| Latin America | 55% | 59% | 61% | 63% |
| Middle East and Africa | 57% | 59% | 58% | 59% |
| North America | 17% | 19% | 19% | 21% |
| Western Europe | 28% | 29% | 32% | 34% |
| Total World Wide | 39% | 43% | 42% | 43% |
| European Union | 29% | 31% | 33% | 35% |
| BRICS Countries* | 64% | 67% | 70% | 71% |

BRIC Countries* are Brazil, Russia, India, South Africa and China.

**Table 1.2:** 2016 BSA piracy study result (retail revenue) ($M)

| S. No | 2015 | 2013 | 2011 | 2009 |
|---|---|---|---|---|
| India | $2,684 | $2,911 | $2,930 | $2,003 |
| Total AP | $19,064 | $ 21,041 | $20,998 | $16,544 |
| Central and Eastern Europe | $3,136 | $5,318 | $6,133 | $4,673 |
| Latin America | $5,787 | $8,422 | $7,459 | $6,210 |
| Middle East and Africa | $3,696 | $4,309 | $4,159 | $2,887 |
| North America | $10,016 | $10,853 | $10,958 | $9,379 |
| Western Europe | $10,543 | $12,766 | $13,749 | $11,750 |
| Total World Wide | $52,242 | $62,709 | $63,656 | $51,443 |
| European Union | $11,060 | $13,486 | $14,433 | $12,469 |
| BRICS Countries* | $14,452 | $17,187 | $17,907 | $14,453 |

BRIC Countries* are Brazil, Russia, India, South Africa and China.

However, various methods and tools for clone detection have been interpreted to some tools, which are included in an estimated evaluation and several attempts have been made to evaluate and assess abundant state-of-the-art equipment.

To demonstrate scalability collisions created by maiden use-cases, we review a retail banking application system carried by "Tata Consultancy Services (TCS)", which is heavily employed by many banks (with various codebase). TCS actuates a comprehensive codebase for the entire banks. Therefore, the primary incentives being intricate due to the industrial losses which arise due to replicated endeavors in (i) assigning common aspects and (ii) maintain general issues exclusively. To get started, due to being one of the intimidations made by the organization, such a large codebase had to classify the block of the general code. Each codebase is completed in millions of lines of code which spreads beyond just a few thousand LOC of COBOL programs, most of which are LOCs of about 100K . The absolute survey of this case study is extant in [10-11]. Hence, the main intention of Tata Consultancy Services (TCS) is to identify the thorough

similar segment of code across entire codebases to support the faction. Please note that while we maintain it with the current expansion, there are many tools and approaches to identifying the code clone. The preceding real life situation is targeted only on how important it is to increase the study to be essential in the vicinity of the business.

Furthermore, a little experimental valuation has been performed by measuring the performances of these tools. The existing study demonstrates that there are a lot of facets that could influence the authenticity of the results of these assessments [12]. To overcome the effect of this facet; there is a need to introduce an outline of the mutation operators-based nomenclature of distant kinds of software clones so that by proposed classification several clones can be generated. The software clone is a result of some alterations in the source code, and these modifications are carried out by some peculiar conventions which are called mutant operators and the emerged faulty versions are called mutation generation [4, 5]. The flawed variants can be generated by some amending transformation in the source code, and these mutation operators used to create probable flaws in code clones to transform the original program [13-14]. The proposed nomenclature used to create a scenario which can present the survey of software clone detection methods and tools analytically. The code clones frequently arises in source code due to replication from one segment of source code and thrashing them into other section of code. The procedure of copy-cut-paste is familiar as code cloning and generated duplicated code is known as clone of code. An error detected in one place of source code necessitates alterations in whole copied fragments of source code. Various scientists have reported 66% [15-19] source codes copied due to copying/pasting. The problem with the duplicated code is that it increases the efforts need to be done when enhancing the source code. The software clones can be categorized as syntactic and semantic clones can be assorted as type-4 clones (distinct syntactic structure but similar functionality; also known as non-trivial clones) [8-9]. Literature has many software clone detection techniques that can detect Type-1, Type-2, and Type-3 software clones with less effectiveness criteria such as Precision, Recall, Portability, Scalability, Robustness. Hence, there is a need to develop an approach which can detect type-1 to type-3 and non-trivial software clones with effectiveness criteria. However, there is also requires several flawed variants or duplicated code in software clone detection to evaluate software clone detection methods. Although, several methods have been taken by several researchers for generating faulty versions by introducing flaws into the correct program. Moreover, these bugs can be initiated by hand or generated automatically through a flawed variant of program. Usually, an automatically-generated semantic software clones. Syntactic software clone is classified as Type-1 (the same code segment beside some amendments in comments, whitespace etc.),

Type-2 (same code segment except some modifications in variables, names, identifiers etc.), Type-3 (Delete similar code pieces with additions of changes such as add, delete statement). The variant as a result of applying some editing activities in the source code. Thus, there is a need to develop a tool for injecting software clones in the source code automatically.

## 1.2 PROBLEM STATEMENT

Expansion of Digital Technology expands the risk of illegal replication of software. Growing piracy rates have given serious risk to the developers towards the expansion of various code security methods. However, there is several code protection methods have been evolved against several malicious assaults as Software Piracy, Reverse Engineering, and tampering. The security methods against these assaults are software watermarking, code obfuscation and tamper-proofing [4-5]. Though, code obfuscation used to transform a code in such a manner which is the thornier for an attacker to comprehend the functionality of the code. The main intention for protecting software through code transformation occurs from the three forms of reverse engineering assaults which are stated below:

- Static analysis attack: statically analysis of data.
- Dynamic analysis attack: to check code on distinct inputs.
- Code clone detection attack: to identify copied code [6-8].

The code clone detection has become a prominent research issue in software engineering, over the past few years due to the replication of code segments in the source code. Reusing source code fragments from one location and paste them into another place with significant alteration is code clone and the pasted segment is copied code. The software clones can be assorted by their aspects as syntactic similarity-based software clones or functionality similarity-based software clones. The syntactic similarity-based clones are identical in their structure as well as in functionality. The textual/syntactic similarity-based clones can be classified as type-1 to type-3, and semantic or functional similarity- based clones are type-4 which is illustrated below:

- Type-1 clone: Identical code segment excluding few changes in whitespace and comments, etc.
- Type-2 clone: Syntactically code fragments of similar type except for modifications in function names or variables.
- Type-3 clone: Identical code segments with some amendments as statements inserted and deleted.

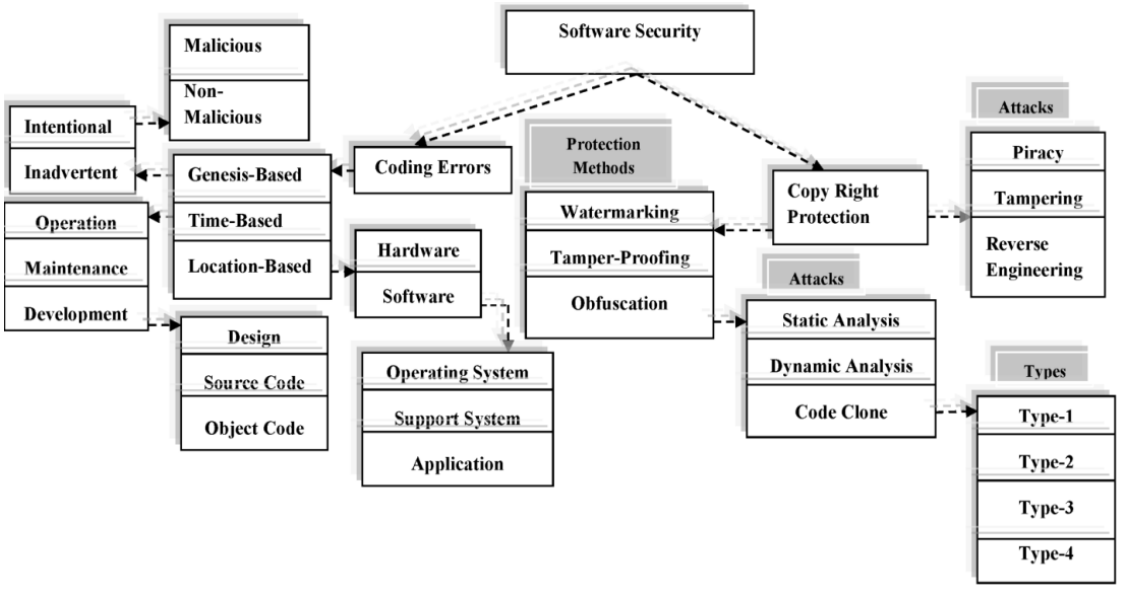



**Figure 1.1:** Graphical representation of problem formulation

- Type-4 clone: Similar functionality distinct structure [20-22]. However, there are various methods for detection of code clone have been evolved for the type-1, type-2 and type-3 code clones, and in each approach, there are its pros and con. But none of the methods identify Type-1, Type-2 or Type-3 code clones with various potency standards as scalability, portability, robustness, precision, recall. Thus, it is necessitating proposing a method, which can identify Type-1 to Type-3 clones with an efficacy benchmark.

## 1.3 RESEARCH OBJECTIVE

The primary research objective of this work is to provide secure software against the three attacks, code clone, static and dynamic analysis on obfuscation. This research will look at whether the anticipated purposed work is accomplished at each level and efficacy of proposed methods is improved in considerable way.

## 1.4 CONTRIBUTION OF THE THESIS

Therefore, the augmentations of this thesis are five-fold as follows.

1. First, ensuring the security of software using obfuscation (data and layout). Further, secure obfuscation used semantic code clones against static and dynamic analysis attacks.
2. Second, Identification of Type-1 to Type-3 code clones. Type-1 code clone detection has been detected using the hybrid method while type-2 clones have been detected using the

DAG approaches. The type-3 code clone detection has been detected using CFG and RFG from the source code.

3. Third, to assess the execution of code detection tools and techniques concerning software metrics as precision, recall, portability, scalability, and robustness. Additional, exposure tools and approaches have been evaluated by created test cases.
4. Fourth, to trounce the ambiguity in clone conception, we have anticipated mutation operators-based editing taxonomy for code clone creation.
5. Fifth, to assess and evaluate techniques and tools for detection of code clone, we have also proposed a mutation operator-based framework for injecting code clones in software code automatically and competently measure Precision and Recall of detection tools of code clone for distinct forms of code clones, which are created using editing taxonomy.

## 1.5 ORGANIZATION OF THE THESIS

Chapter 1: Illustrates the main motivation of this study hitch in obfuscation, code clone detection and study accompanied by the augmentations of this hypothesis. The remaining structures as follows for this thesis:

Chapter 2: This chapter is a synthesis of a additional impetus to this hypothesis, few background abstractions and literature survey about the software security.

Chapter 3: This chapter presents a basic introduction of code clones, related work, and open research issues in code clones.

Chapter 4: In this chapter we anticipated a novel obfuscation approach for code protection. Moreover, to protect obfuscation against static and dynamic analysis using non-trivial code clones.

Chapter 5: The aim of this chapter detection of code clones from type-I to type-3 in the source code to diminish maintenance cost, increase performance speed. etc. Therefore, we proposed a hybrid technique for the detection of Type-I code clones while on the other side we used Directed Acyclic Graph method for the detection of type-2 code clones. The trivial code clones (type-3) detected using CFG and RFG method.

Chapter 6: This chapter provides a performance valuation of existing code clone detection tools and techniques by software metrics. Although, we evaluate detection tools and techniques by

generated test cases. These test cases are generated using mutation operator-based editing taxonomy which is shown in the form of an illustration.

Chapter 7: This chapter, foremost presents the definitional ambiguity of code clones. Thus, we proposed a mutation operator-based expurgation nomenclature for clone generation in the form of an illustration. Further, by using this classification, we proposed a hypothetical scenario for evaluating code clone detection tools and techniques. To measure the accuracy of code clone detection tools a mutation operator-based automatic framework has been proposed which is used to inject and detect code clones.

Chapter 8: At last, this chapter summarized the thesis accompanying thesis summary concluding remarks, the contribution of the work done, and some future directions for the research.

# CHAPTER 2

# BACKGROUND AND PRELIMANARIES

This chapter presents background and related work on software security. We eventuate with section 2.1, which provides a brief introduction about software security. In section 2.2 we present software security techniques and types of software security attacks are discussed into section 2.3. Taxonomy of protection methods against attacks are discussed in section 2.4. Watermarking's property and protection techniques are conversed in section 2.5. In section 2.6 we present a succinct about tampering, their types and types of attacks. Obfuscation is conversed in section 2.7. Meanwhile, section 2.8 concludes the entire chapter

## 2.1 INTRODUCTION

Software security means to protect code from unauthorized alteration as well as it is the design of engineering software. The objective of software security is it continuous functions correctly under malicious assault. One of the key aspects of software security is stenography, which is a cryptography contract and analyzes how to secretly transmit data. Security of Software has two types named as coding errors and Software Copyright Protection. Intellectual Property Rights (IPR) is a two-player game between two attackers: Programmers (Xs) aimed at securing code from attack, and reverse engineer (Y) whose attempt is to evaluate the code and adapt it into a form that is facile to study and comprehend. It is not essential for B to translate the code back to which is close to X's original source text. Thus, it is necessary that the reverse engineer code should be clarified by Y and it might not be feasible for X to protect the overall application from Y[4-7]. So Software Piracy, reverse Engineering and tampering are three types of intellectual property attacks, which is depicted below in Figure 2.1.

In Figure 2.1(a), B has legitimately buy code from A, makes replica of an application and illicitly sells them to un suspicious regulars. In Figure 1.1 (b), B has bought an application from A and reuses one of module M from A's code and reiterate it in own code. In Figure 2.1 (c), Media content from the digital container of A has been extracted by B. A encompasses of digital media content and text that transmitted an assertive quantity of e-money to A's bank account whenever media is played. B may try to interfere, together the digital container, to get

the media-content or modify the C so that the payment can be made less for the media play. In the latter case, B can sell it again to a third party or enjoy that content continuously free [4].

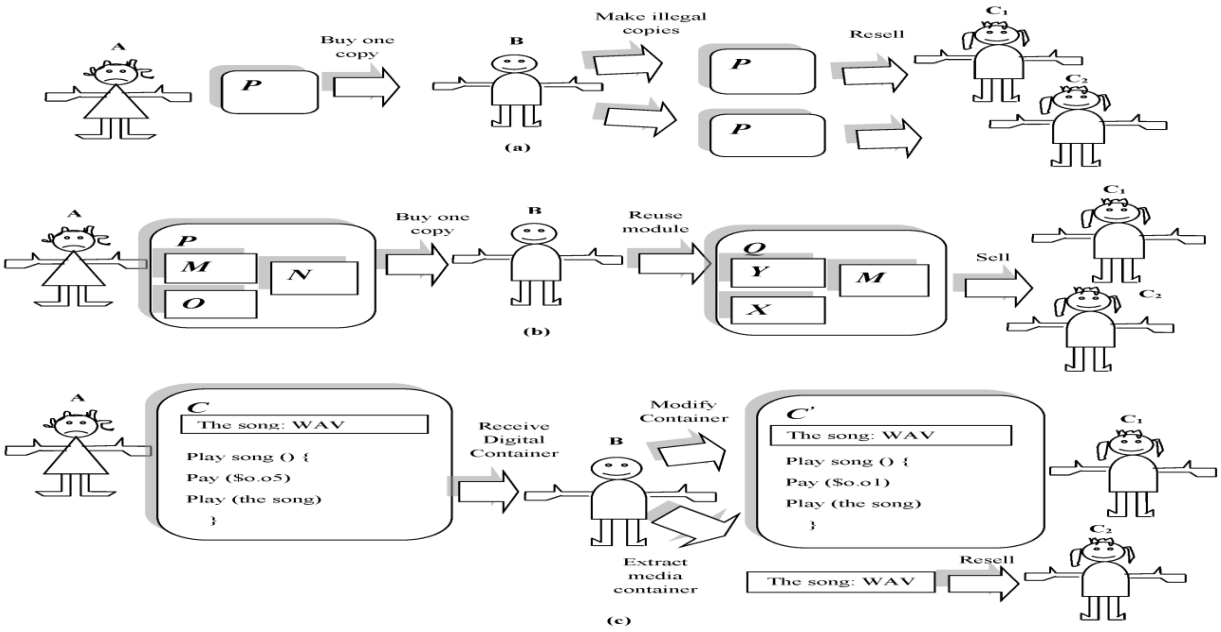



**Figure 2**.1**:** Attacks stand opposite to Software Intellectual Property (a) Software Piracy attack (b) Malicious Reverse Engineering attack, and (c) Tampering attack.

Obfuscation, watermarking, and tamper- proofing are the most prominent methods for the copy right protection of software [4-7]. The aim of obfuscation to make a code compact for an attacker without changing its functionality. Tamper-proofing means to protect software from unauthorized modifications. In software watermarking a signature is inserted in the code to demonstrate the rights of software. However, software fingerprinting is a technique as software watermarking which compasses an exclusive client ID number into every dispensed replica of an application to assist execution of copyright violators [4].

Figure 2.2 depicts copyright protection's methods (a) A embeds a watermark in a program P by a surreptitious key K at number 1. B filches a replica of P's and C extorts its watermark by K to confirm that P's is possessed by A at 2. (b) Especially the special "tamper-proofing code", called 'T', used by 'A' to protect a secret 'S', if the 'S' has been altered by 'B'. and in (c), A makes the program harder for B by obfuscating transformations, while maintaining its semantics.

## 2.2 SOFTWARE SECURITY

IEEE defined security as: (1) security of information and data so that illegal persons cannot alter it. However, the authorized persons/systems can access without any modification [23], (2) all aspects related to defining, achieving, and maintaining, integrity, confidentiality, non-repudiation, availability, accountability, reliability and authenticity of a system [24].

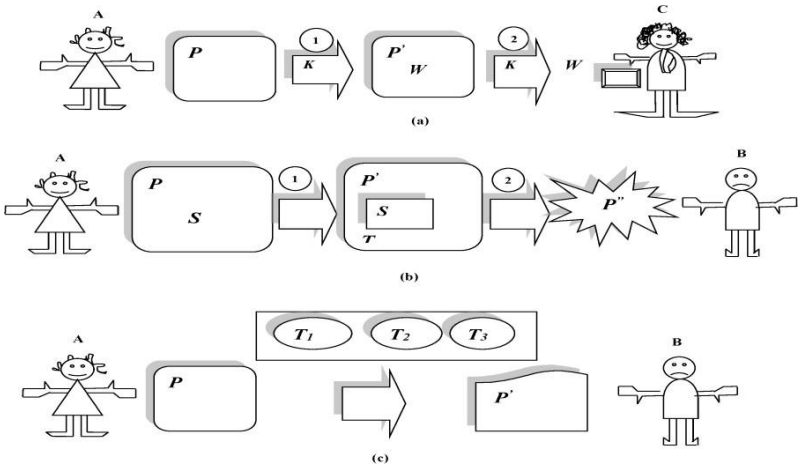



**Figure 2.2:** Protection techniques against software intellectual property attacks, (a) watermarking, (b) tamper-proofing, (c) obfuscation

Security is the protection of valuable information assets from an unauthorized access [4, 6] and it has three components requirements, policy and mechanisms. Requirements define security goals and policy defines the meaning of security. Mechanisms enforce policy.

The security concepts can be classified i) asset refers to information that has value to an organization. ii) Stakeholder is a person who places a particular value on asset. iii) Security objective means a statement of intent to counter threats and satisfy identified security needs. iv) Threat is a potential that cause for harm of an asset. v) Attack denotes an action intended to violate the security of information. vi) Attacker represents an entity that carries out attacks. vii) Vulnerability is a loop hole that could be exploited to gain unauthorized access of an asset. viii) Countermeasure an action that is taken to protect information from threat. Risk means a Probability of damage

## 2.3 TYPES OF SOFTWARE SECURITY ATTACKS

There are three attacks exist in the literature for copyright protection against a program named as P. These assaults can be defined as analysis (reverse engineering), tampering and distribution [4-7]. Although, software is prostrate to many security assaults such as reverse engineering (to comprehend the code functionality), surreptitiousness of code IP, and varying codes functionality by tampering. The piracy means to illegal copying of software. Moreover, these assault can be broadly categorized into three types namely as software piracy, reverse engineering, and tampering [4-5].

## 2.4 CLASSIFICATION OF COPY RIGHT PROTECTION METHODS

Computer security can be categorized into three types as database security, network security, and software security [25]. The "data security" is related to the integrity and confidentiality of information in permeation and repository from unauthorized users while the network security protects the network resources, devices, as well as services. The software security to secure software from unauthorized access, modification, and tampering [26]. There are two types of methods for protecting the Intellectual property of software which is depicted below in Figure 2.3.

Figure 2.3 depicts classification of computer security as well as software protection techniques. Intellectual property based protection methods are two types such legal method and technical method. The legal method related to all probable legal laws and it makes legal actions against unauthorized users. Further, the copyright, patent, registration and license are categorized into the legal laws. The intellectual properties rights (IPR) secure the programmer's right against

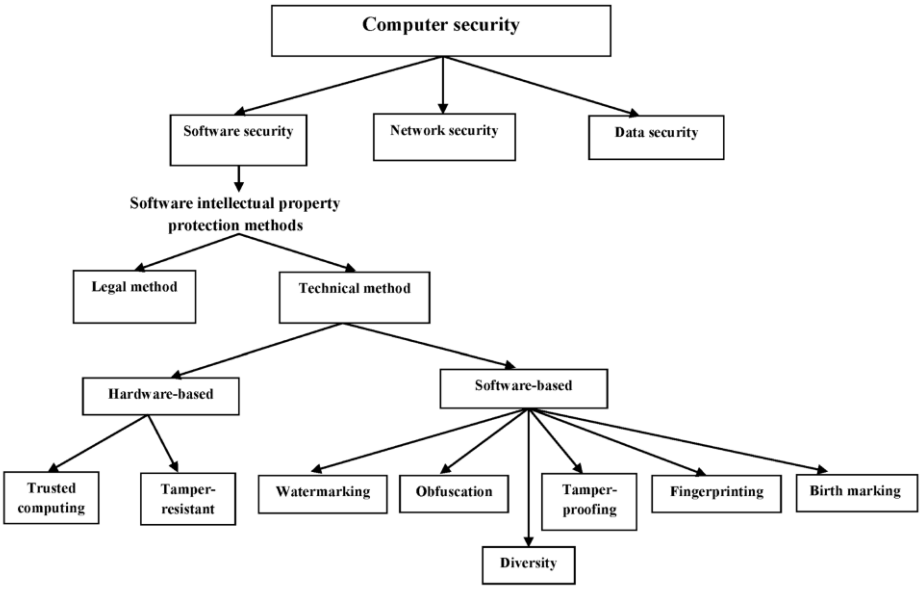



**Figure 2.3:** Classification of intellectual property protection methods

copy /reproduce, distribute and publicly perform the work. Although, IPR rules protects a computer program only it doesn't not protect the methods and algorithms within the program. The other IPR method named as technical method which can be categorized as a) hardware-based protection methods and software protection method. The hardware-based again can be two types as two types as trusted computing (To secure data in hardware by encryption and encryption using RSA 2048 algorithm) and tamper resistant( software and facts are actually secured from assaults)[27]. The software-based method is a type of technical method. The

software-based methods can be classified as data encryption, antidisassebly, and code obfuscation [5, 8, 29, 30]. The main aim of code obfuscation to secure code from reverse engineering assaults. Further, the code obfuscation is semantic preserving transformation that preserves functionality of code and generates the harder code for an adversary. Software watermarking [28-30] refers to a mechanism in which authors proves their ownership of the source code. The tamper-proofing [4, 5, 29, 30] is used to secure the source code from tampering assault. Although, tampering assaults is also secured by parity bit. Finger printing and birth marking [5-6], to detect and trace illegal distribution. Software diversity [29] provides protections by different variants of software which are functionally similar the primary leverages of software-based protection techniques are the low-priced and litheness.

The software watermarking technique embedded an unique identifier into the text of software, to discourage piracy attack [4, 5, 28]. Moreover, such secret identifier used to assert the property rights of the source code. However, software watermarking states the facts of a piracy occurrence; it doesn't avert code from piracy. There are two types of software watermarking subsystem namely as embedding subsystem and extracting subsystem. The intent of embedding sub system [28] to embed watermarks in code while extracting subsystem intends to extract watermarks from watermarked code which are shown in Figure 2.4.

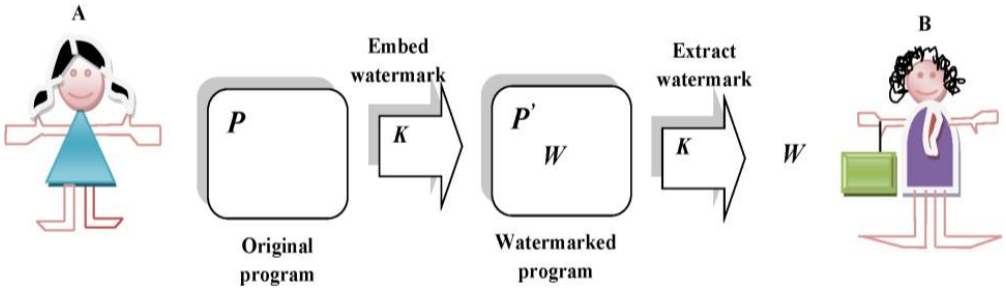



**Figure 2.4:** Watermarking embedding and extracting process.

The watermark insertion and extraction process is illustrated in Fig 7. A insert a W (the set of watermarks) into a program P (denotes the set of programs) using secret key K and gets a watermarked program P'. B extract watermark using the same secret key on P' and gets original program P.

## 2.5 PROPERTY AND PROTECTION TECHNIQUES OF WATERMARKING

Software watermarking method has some security approaches and property, which are shown in Figure 2.5[1, 2].Figure 2.5 (a) shows the watermarking properties as resilience; signify the facility to withstand various forms of dewater marking assault. Stealth, quantifies the dissimilarity between the several kinds of instructions which is used to program computations as well as to insert a watermark. Credibility, while minimizing the probability of coincidence. Data-rate enlarges the size of the message which can be embedded. Perceptual invisibility, In order to make detection difficult a watermark should demonstrate the similar assets as code around it. Part protection, orderly to secure entire elements of the software, a watermark should be distributed throughout the software. Overhead means a watermark insertion and extraction process should not be so costly as well as slight affect on the execution of the code. Figure2.5 (b) depicts the security methods of watermarking such as fingerprinting; the untrusted system single out by it on which the trusted code was executed while the watermark identifies only the software authors. The objective of TCPA (the trusted computing platform alliance) subsystem is to use public- key cryptography as well as facilitate public-key framework to allocate a consistent uniqueness to every system. The obfuscation makes code thornier for an attacker to locate the watermark and tamper-proofing harder to modify. The functional based watermarks are basically four types such as 1) “prevention marks”, to avert illegal employs of code,2) “assertion marks” refers to proves the ownership on the applications,3) permission marks permit a partial alteration or replica process to the application , and 4) the affirmation marks assure code’s authenticity. The embedded/extracting technique based watermarking are two types as static and dynamic. 1) Static technique to insert unique identifier in the text or the program’s codes before the execution of program. Further, static watermark is divided in two parts such as: a) “code watermark” (to insert watermark into the code part of an application) and b) “data watermark” (embedded directly into the data vicinity of a code).

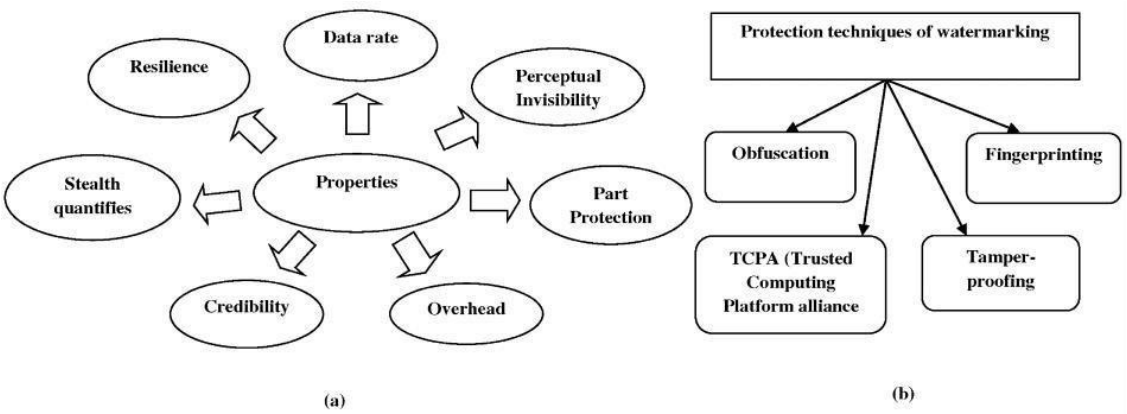



**Figure 2.5:** Various watermarking (a) property and (b) protection techniques

### 2.5.1 Taxonomy of Watermarking

Figure 2.6 demonstrates the taxonomy of watermarking which is categorized by four on the four aspects. These aspects can be described as: 1) functional goals basis, 2) embedded/extracting techniques based, 3) human perception, and 4) according to watermark extractor dynamic watermark inserts watermark at the execution state of software object. The dynamic software watermarks are basically three types [4, 32] such as a) the easter egg watermark, whenever a unique input string is inserted, it executes various act that is instantaneously observable by the user in the form of a copyright message or an unanticipated picture. Although, the main constraint of easter egg watermarks is it easily detectable. While the dynamic execution trace watermark doesn't formed any outcome like easter egg watermark. In addition to this, the watermark is inserted within the trace of the application as it is being execute with the special input, and c) dynamic data structure watermark is not generate any output like execution trace watermark. The human perception based watermarks are two types such as 1) visible watermark, and2) invisible watermark. The "visible watermark will create a "coherent image like a logo". It illustrates the presence of visible watermarks in the application while in Invisible software watermarks legible image is not shows to the "end-user" but can be extort by several algorithm. Further, invisible watermark is classified such as: a) robust watermark (to avert illegal uses in organization that formulate communal asserts to software rights), and b) "fragile watermark" [28] (to be eradicated when the code has been altered). The watermark extractor can be two types named as blind watermark and informed watermarks. In blind software watermarking, only watermarked program is given to the extractor and "the watermark is used as its input while in informed software watermarking the watermarked program" as well as inserted watermark is given to the extractor.

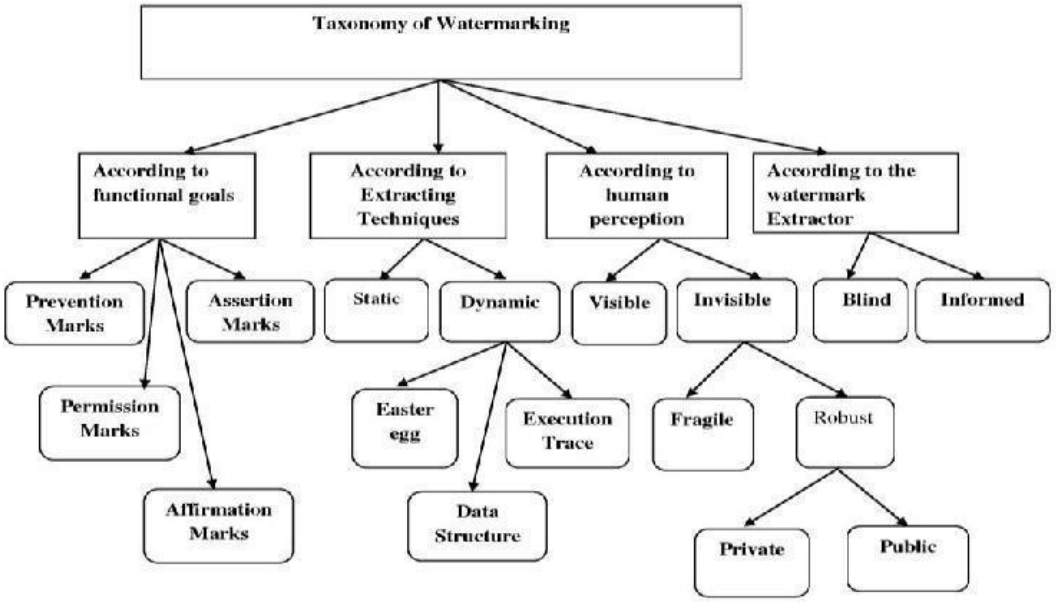



**Figure 2.6:** Taxonomy of software watermarking

### 2.5.2 Classification of Watermarking Attacks

Malicious Host and Malicious Client attacks are the two types of attack that occur on Software. However, Malicious Host permit Software- Watermarking to protect the software Figure 2.7 depicts four types of watermark attacks as additive attacks, subtractive attacks, distortive attacks, and recognition attacks [4, 31, 32].

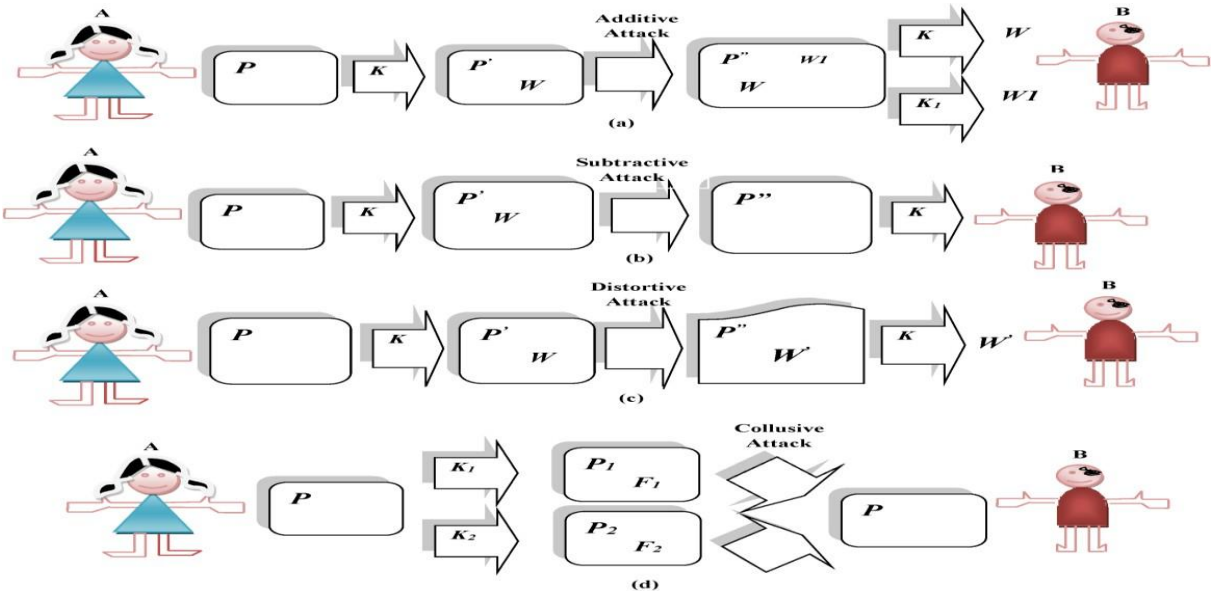



**Figure 2.7:** Attacks against software watermarking, (a) additive attack, (b) subtractive attack (c) distortive attack, and (d) collusive attack.

Figure 2.7 shows the software watermarking attacks. (a), B embeds new watermark in to watermarked program P' and A (proprietor of the software) cannot prove their property rights on the code. (b), B eradicates the watermark from the software without impacting the functionality of the application. (c), the watermarked code is altered by the B so that the watermark cannot be extorted by the A without changing software's usability, and (d) , B acquires various facsimiles/copy of A's code every with distinct fingerprint F1,F2.B easily locate the fingerprint and can remove them by evaluating distinct replication of code . Moreover, the removal assault, to eradicate entire watermark data from the watermarked code without negotiation the protection of the watermarking algorithm, and "recognition attack to adapt the watermark detector, or its inputs, so that it gives a ambiguous consequence".

## 2.6 TAMPERING

Tampering is a technique that makes unauthorized modification/alteration in program/code and tamper-proofing is a protection method to prevent such tampering attack. The distinction in "tamper-proofing and obfuscation" is that a code which is complicated to comprehend because of obfuscated code which is also a thornier for an attacker to alter. There are various tampering approaches proposed in the literature named as reverse-engineering on chips, glitch, micro probing, power analysis and cipher instruction search assault [33].

### 2.6.1 Categorizations of Tampering

Tampering can be categorized into types as "static and dynamic tampering". The "static tampering" related to modify a static binary image". In static tampering assault to digitizing a fissure and employing it on the stored binary. Further, pretentious that program is not encumbered in memory and customized there [6]. In "dynamic tampering attack", to change the code at run time. The dynamic tampering attack is generally carried out by hand as well as it has similarities to software debugging [6].

### 2.6.2 Types of Tampering Attacks

Tamper-proofing system comply two assaults as first it observes the code as well as the execution environment. Second, when it is actuated that tampering had ensued then a response mechanism taken over and revived in appropriate way. Thus, the objective of "tamper-resistant" code to thwart interfering of a code using: (1) identified unauthorized modification, and (2) retorting in case of altering.

Figure 2.8 shows the classification of protection techniques against tampering Anti disassembling techniques (Code Encryption, Nebelbombing, Virtual machine) basically used to delay or prevent analysis of code. Anti-debugging techniques which are two types hardware specific includes trapping, (to prevent interruption in the execution of the program), processor errors, port and the other is platform specific (e.g. "stack smashing" is causing a stack overflow in a computer application or operating system). Code-encryption and obfuscation are the Anti-modification techniques that can be categorized as code encryption integrity checks and anti-decompilation methods.

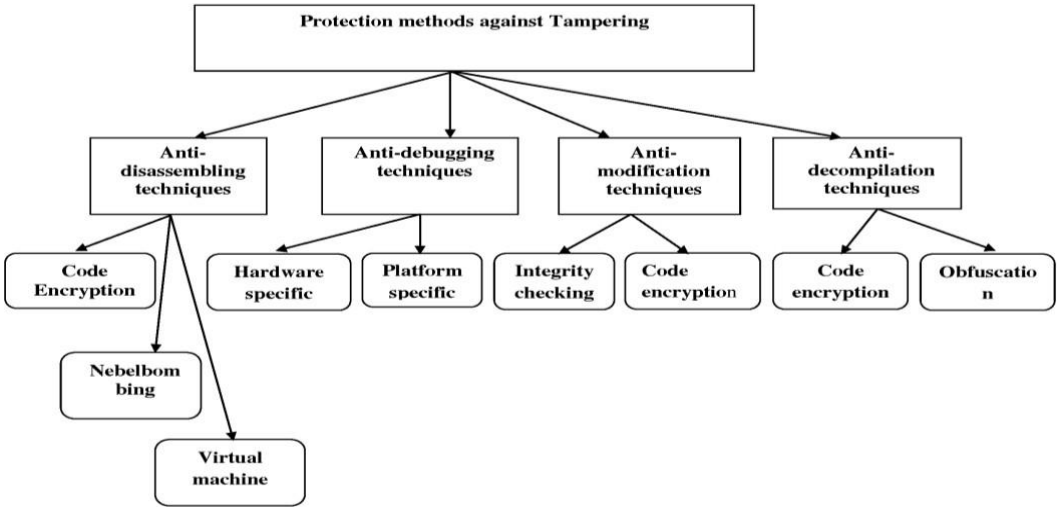



**Figure 2.8:** Taxonomy of tampering protection techniques

## 2.7 OBFUSCATION

An obfuscator 'O' is an efficient and probabilistic compiler that takes as input a program 'P' and transforms that into a program 'O(P)', which is obfuscated form of 'P' and has same input-output or functionality as the original 'P'[34].

### 2.7.1 Merits and Property of Obfuscation

Collberg et al. [5,7] presented some evaluation properties of obfuscation. In Figure 2.9 various features of obfuscation have been discussed. There are some the obfuscation properties which are depicted in Figure 2.9 (a) The reader impeded by transmuted code has an extent namely as potency. Flexibility, the degree to which transmogrification 'T' can prevent an automated de-obfuscation. Due to alterations, Cost and performance overhead will added [9, 36]. Characterized transformed code from source code is Stealth. [5]. Figure 2.9 (b) demonstrates some advantage and disadvantages of obfuscation. The protection can be defined as to protect source code against static and dynamic analysis assaults. The main aim of protection is to make

code compact for an attacker, and it necessitates more endeavors to comprehend the code. Diversity is defined as to generate clear illustration of the original code which is structurally diverse but functionally similar [106]. Although, cost (to transform code each alteration need the additional charge in memory) and security are the main constraints of obfuscations [6].

### 2.7.2 Taxonomy of Reverse Engineering Attacks and Security Techniques

Figure 2.10 exemplify several Reverse engineering assault and security methods. Code clone, static and dynamic analysis attacks are three classes of Reverse Engineering [9, 6] which are shown in Figure 2.10 (a). The functionality of a program can be apprehended with the help of these attacks by an attacker. Static analysis attack, in this assault adversary analysis information for all execution [5].In dynamic analysis attack, an adversary run the code on different set of input and analyzes the result of the code by creating execution traces as well as reveal factual trails elected for code implementation. In software clone assault, an attacker adversary identifies and eradicates copied segment present in the source code to understand the semantic structure of the code. Figure 2.10 (b) shows the Reverse Engineering protection techniques.

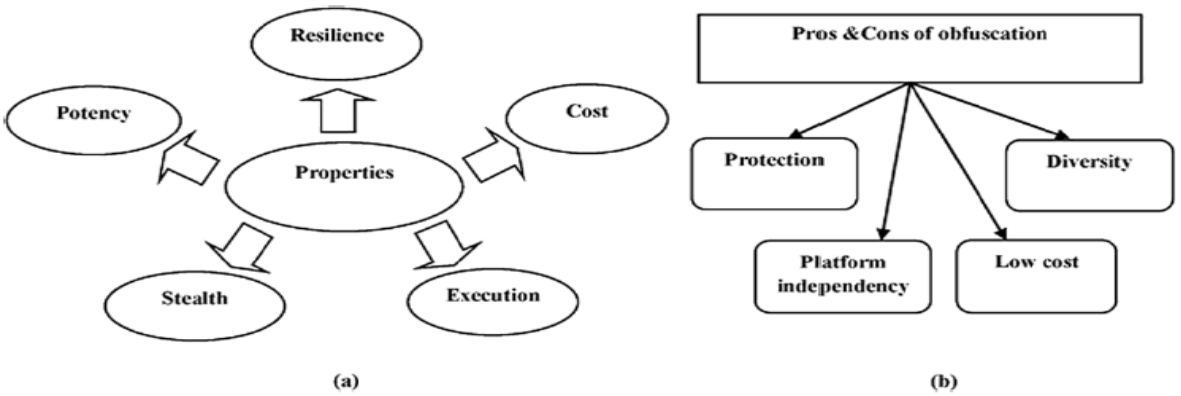



**Figure 2.9:** (a) Property of Obfuscation (b) Pros and Cons of Obfuscation

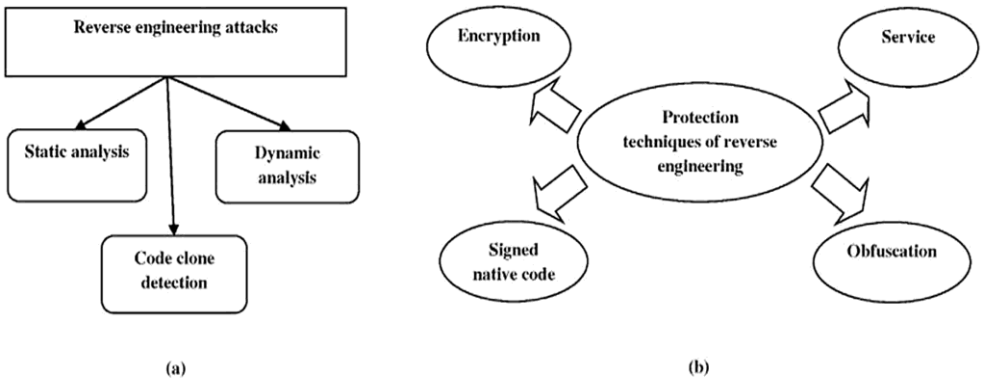



**Figure 2.10:** (a) Types of Reverse engineering attacks, (b) Protection techniques for Reverse engineering

The security methods against these attacks can be classified as encryption, signed native code, obfuscation, and services. Encryption is the important in all counteractions against malicious reverse engineering, which has on demand encryption /and decryption and decrypted part is executed and instantly re-encrypted at runtime. However, due to the many encryption and decryption call, encryption system has many implications regarding performance. [7, 97]. The other safest protection approach against malicious reverse engineering is service. In service method users haven't access to the application but they can use it after paying small amount. Additionally, an adversary unable to reverse engineer as well as they haven't physical avenue to the software. Just-in-time compiler translates the Java byte code in signed native code approach which is more complicated for an adversary. In the signed native code there is no authentication on user's system before implementation, like Java byte code which is the main constraints of signed native code [7]. The other one of the most popular approach of reverse engineering approach named as obfuscation which makes code compact for an attacker. In addition to this, another defense techniques of reverse engineering which exists in literature namely as anti-disassembly and anti-debugging [4].

### 2.7.3 Taxonomy of Obfuscation

There are many protection techniques against malicious reverse engineering which are shown in Figure 2.11(a), and Figure 2.11(b) presents various kinds of obfuscation properties. However, there are several different classes of obfuscation techniques which are illustrated in Figure 2.11 (c) These techniques can be categorized by types of data it targets, and the function it executes. The code transformation can be mainly divided into four types and their subtypes, which are depicted in Figure 2.11 (d-g) as 1) data, 2) layout 3) control and 4) preventive transformation of obfuscation. Transformation of code can be classified into data, design, control, and layout obfuscation [35]. Due to the diversity of the data formation of the code, the obfuscation of the data to disrupt the opponent by extracting information from the source code. Storage, Aggregation and Ordering are three classes Data obfuscation [8, 36]. The lexical obfuscation entailed to change the layout of the program by using a assortment of amendments in source code as renaming variables, erasing comments, eradicating debugging data. The control obfuscation used to change the control stream of the code.

Furthermore, computation, aggregation, ordering and control flow flattening are various methods available to change the control flow of the source code. [8, 35, 36]. The de-obfuscation techniques can be become more difficult by the preventive obfuscation. Preventive obfuscation

has two types as targeted and inherent [7, 8, 97]. The design obfuscation makes the layout of code compact for an adversary by changing the structure of the program. Design obfuscation can be accomplished by using splitting classes, hiding type information, and merging classes. Moreover, another type of obfuscation is language level. Language level obfuscation has three categories, one intermediate (assembly language, byte code) level, second source code (high-level language such as Java) level [6], and last binary level [35].

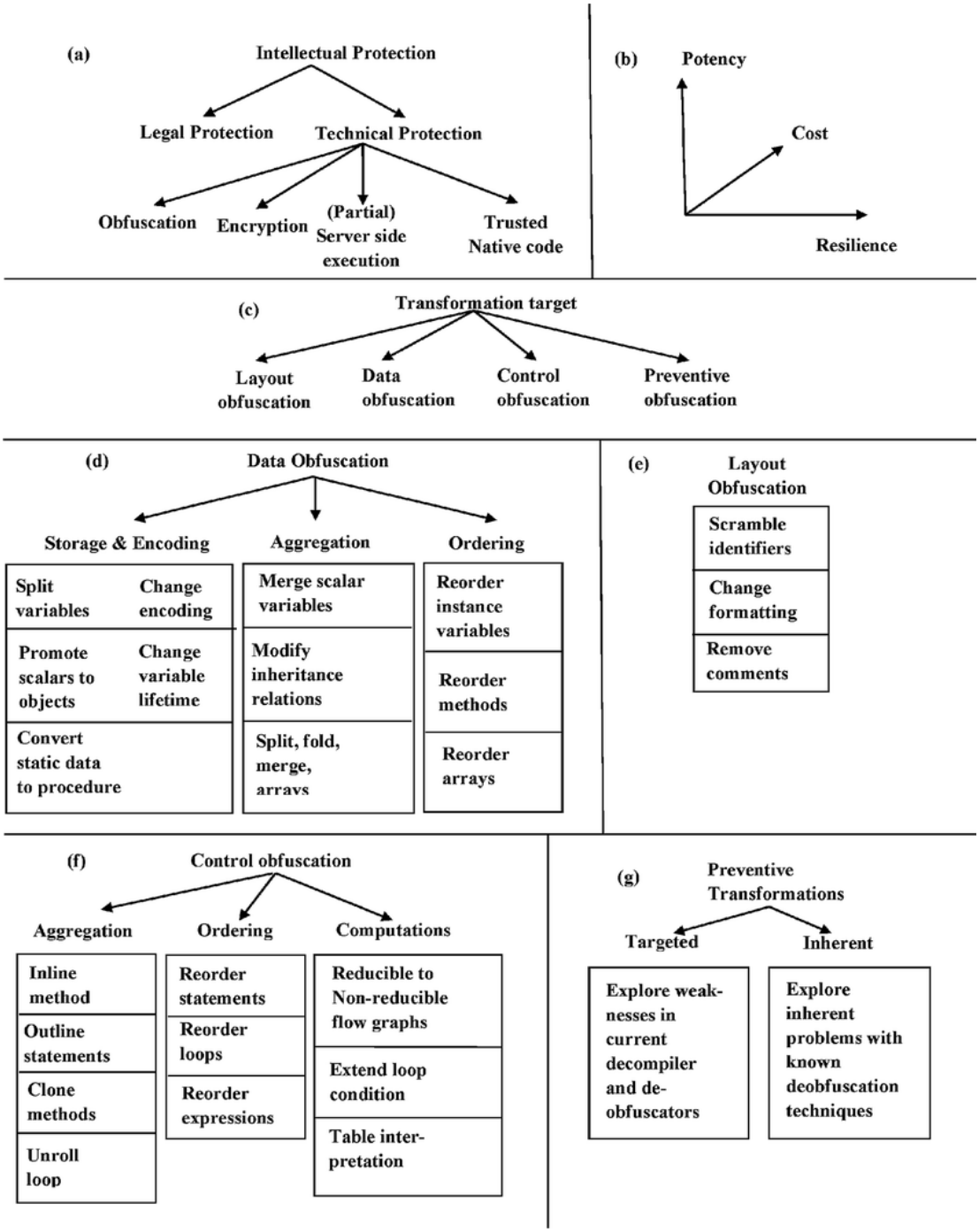



**Figure 2.11:** Taxonomy of (a) Types of protection against malicious reverse engineering. (b) The features of an obfuscating transformation, (c) Information pointed by an obfuscating

transformation. (d) Lexical transformation. (e) Data transformation. (1) Control obfuscations, and (g) Preventive obfuscations

### 2.7.4 Open Issues in Obfuscation

There are several open research issues in obfuscation such as performance overhead, long term security and code clone detection. Performance overhead, related to memory usage and extra cost. Obfuscation introduces extra cost in perceptive of "execution time and memory usage" due to transformation because execution time essential to perform the transformation of source code. The other research issue is no long-term security, transformation does not protect code completely it makes only code analysis harder for an adversary, not impractical. The main obscurity of an efficient "code obfuscator" is to assure protection, i.e. to determine that the software protection cannot be compromised using any algorithm within realistic time. The obfuscation methods and tools depend on the informal notion of security and thus can't be envisaged as conclusive protected [37]. The third research issue in obfuscation is related to non-trivial code clone detection in Obfuscation. Process of reusing modified source code is called cloning and the duplicated code, code clone [20, 21, 111]. Further, the non-trivial code clone is an example of type- 4 code clones. The semantic code have similar functionality but diverse formation. However, if code clone has some advantage as by injecting duplicated code to make static analysis harder for an adversary but on the other side it has some limitations. As code clone have bad impact on design it leads to bug propagation [20, 21]. The authors [9] have mentioned non-trivial code clone cannot be detected using any existing static analysis methods**.**

## 2.8 CONCLUSION

As per the BSA report [1] every year a huge amount has been spent to prevent software piracy. Hence, the protection from piracy is an emerging issue in present scenario. Further, software piracy is a type of software security's attack. Thus, the primary aim of this chapter is to presents a thorough survey on software security. The software security attacks can be categorized as software piracy (the illegal copying or resale of applications), reverse engineering (to reuse or extract model for their own purpose) tampering (unauthorized modification). Further, the protection methods against these attacks can be classified as watermarking, obfuscation and tamper-proofing. In addition to this, one of the significant techniques of software security named as obfuscation is presented with their research gaps in today's scenario.

# CHAPTER 3

# CODE CLONES

This chapter presents motivation background and related work on software clones. We introduced software clones with abstract definition and their classification in section 3.1. In section 3.2 we will discuss advantage and disadvantages of the software clones and what are the roots causes of software clones are discussed in section 3.3. Various detection tools and methods are also explained in section 3.4. Section 3.5 illustrates some open research opportunity in code clone detection and at last, section 3.6 concludes the chapter.

## 3.1 SOFTWARE CLONE TERMINOLOGY

Figure 3.1 shows the attribute-based categorization of code clones. The attribute-based taxonomy of code clones Classification of clone is valuable for optimizing the detection and re-engineering approaches. Based on the clone taxonomy, we have increased the most extrusive forms of clones, which happen at Re-engineering time. Similarities among clones, location in source code and Refactoring opportunities with detected clones are the categories of code clones based on attribute [21]."

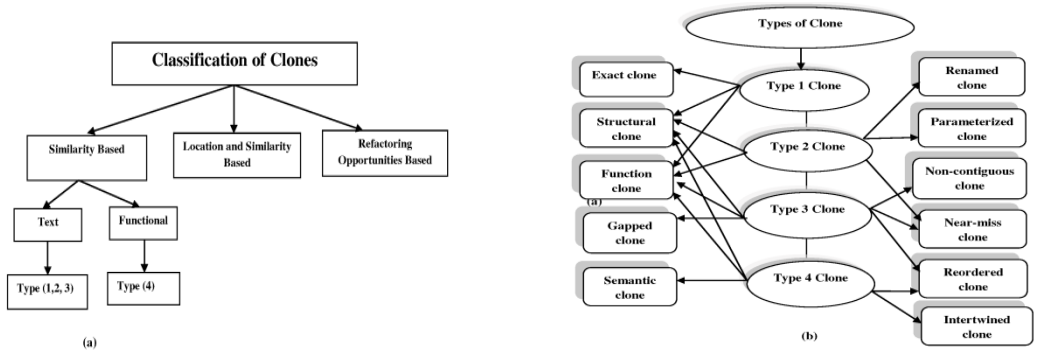



**Figure 3.1:** Taxonomy of code-clones based on (a) attributes and (b) similarity.

Figure 3.1(a) Software clones are two types on the basis of their similarity as: 1) text content-based similarity, and 2) On the basic of functionalities of source code, two-code-fragments become identical. Though, syntactically similarity-based clones can be assorted into Type-1, Type-2 and Type-3. Semantically equivalents clones are type-4 clones as different structure but identical functionality). After replication, developer distorted syntactic elements, which evaluates by this classification. For example, high-similarity clones include those methods that

are identical except for names or methods, but for types of parameters. However, this kind of information generally indicates a refactoring.

There are four types of code clones and their sub-types, which are depicted in Figure 3.1(b). The textually similar, functional-based clones are classified among type-1 to 3 and type-4, respectively. Further, type-1 clone can be classified as the exact clone, structural and function, Type-2 as renamed, near-miss and parameterized, Type-3 as structural, function, , near-miss, gapped, reorder and non-continuous, and finally Type-4 as structural, function, reordered and intertwined [111].

## 3.2 SOFTWARE CLONE PROS AND CONS

Sometimes, existing code clone in software is deliberately introduced by software developers. Kapser and Godfrey [37, 38] proposed thrash out these issues. Some points are given below:

- This is a quick and immediate way to demonstrate the amends requirements.
- Today's various examples in programming persuades to use of templates.
- If there is a lack of duplication and perception method in a programming language, then this is the only way to increase immediate functionality immediately.
- Overhead of function calls infrequently support code duplication for competency concerns.

**Code-clone's Cons**

- **High maintenance costs**:" Existing literature [40, 41] shows that the duplicate code enhances the preventive and adaptive attempts of the software, highly.
- **Bug propagation**: If a bug found in one code segment then it should be corrected in the entire replicated code segment. Thus, software cloning augments the possibility of bug propagation [42, 43].
- **Bad impact on design:** Code cloning leads to bad impact design as we as it abash the use of refactoring, inheritance, etc. [44, 45].

## 3.3 REASONS OF SOFTWARE CLONE IN SOFTWARE SYSTEM

On the point of maintenance, Copy / cut / paste are the best examples, used to study for bad practices, while many software developers use them. There are some inferences to code cloning mentioned below:

- **Software developer's restraint and time inhibition**: The software is rarely written under idyllic situations. Restrictions of developer's ability and time limits obstruct legitimate software development [46].
- **The difficulty of the system:** The complexity in perceptive large systems simply endorses replication the presented logic and functionality.
- **Language constraints**: Why software developers do copy and paste code, complete an ethnographic survey on it by Kim et al. [46]. They discover that programmers sometimes apply to copy and paste the code because of restrictions in languages. There is a lack of intrinsic support for the code restate in many languages, which leads to replication.
- **The aversion of new code:** Generally the programmers are afraid to carry novel ideas in existing software. They suspect that the introduction of new code may result in long "software development life cycle (SDLC)". Additionally, instead of expanding the new code, it is easy to rewrite the existing code because the new code can lead to new flaws [47, 48].
- **The scarcity of reformation:** Due to the time restriction, Refactoring and abstraction etc. of code are impediment of reorganization for software developer. Normally, after the release of the product, the restructuring gets delayed, which increases the resulting maintenance cost. [49].
- **Templating:** Forking are the same results as other, hope that the development of the code will emerge independently in the short-lived [39]. Functional and structural templates are often used as reuse methods.

## 3. 4 DETECTION METHODS FOR CODE CLONE

The performance evaluation of clone detection approaches accomplished by some basic properties. The detailed description of these properties is mentioned in the following section.

Figure 3.2 illustrates the some basic properties of clone detection approaches as Normalization can be defined as to employ a number of modifications on the source code before actual assessment such as to eliminate whitespace, comments etc. Transformation's results are called source representation. The Comparison Phase used the Granularities for a meticulous system. Further, the other significant property of clone detection technique is related to the comparison algorithm, which plays an imperative role in the exposure of different forms of code clones. Though, the techniques complexity depends on the nature of the comparison algorithms and

kind of transformations which is used by clone detection technique. The clone parallelism characterizes different types of clones that are detected in different ways. Granularity be either fixed or free. Language independence property confirmed the language outbreak of an identification tool.

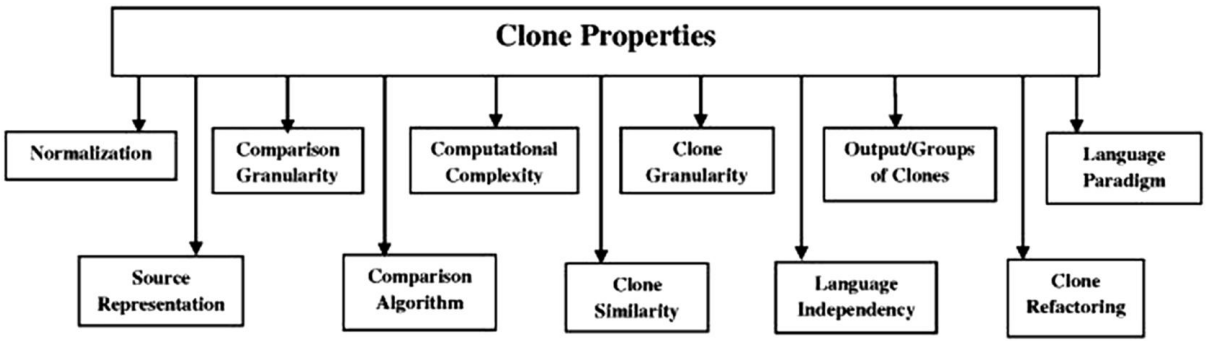



**Figure 3.2:** Classification of code clone attributes.

The output characteristic specifies, what type of output will be in a clone pair or clone or both. Restructuring of existing source code, without changing external behavior, is called refactoring of code. The language paradigm is the programming paradigm, which is targeted for a special method of interest. However, there are several proposed clone detection techniques emerged over the past decades. Detection techniques for code clone are classified into the following types:

### 3.4.1 Text –Based Methodology

This approach considers the source code segment as the text afterwards. Various transformations like whitespace, newline and removing comments etc. are the basic for textually comparisons of two segments of code, to locate the same strings sequences. Various scientists have expected many string / text-based approaches to identify the clones. “Baker” [50,51] include string matching algorithm, based on lexer and line, on token for line with the help of a tool namely as “Dup.” For detection of clone, which has different variable names, he uses special parameter. Meanwhile, this tool has some restrictions such as neither it cannot detect clones written in a distinct manner, nor maintains exploration and navigation among the duplicated code. Subsequently, “Koshke et al.,” [52] used tokens and non-parameterized suffixes to overcome this problem. However, the authors were unable to detect exact and parametric clones, and they could not distinguish among them. In addition to this, the clone (text based tool) proposed by “Koshke et al.,”[52] does not check whether the names of

identifiers have been changed or not changed. “Johnson” [53] used “Fingerprinting Algorithm”, which designed by Karp-Rabin, for detection and compute the fingerprints of a code for an entire substring of the program. The entire code is divided into a set of “sub-string” due to every character in this method subsists of at least one sub-string and then normalization is employed for similarity detection of those sub-strings. However, in this approach fifty lines are matched, resulting reduction in false positive numbers. “Cordy et al.,” [55] used an island grammar technique for detecting syntactic constructs. Furthermore, near-miss code clones detection for web pages written in HTML, also given by the authors [55]. T The proposed method by authors [55] used smallest comparison, which is the main restraints, as well as unable to normalization any code. Ducasse et al., [56] introduced a language independent method based on dynamic pattern matching algorithm.

Due to the cohesiveness of code, this proposed approach could not detect significant clone pledge in language-independent way. A latent semantic indexing-based [57] methodology developed by “Marcuss” [58]. As per the aforesaid detection techniques it is shown that it does not employ code transformation on the “codebase”. One of the recent method introduced by “Ducasse et al.,” [56]. In spite of the fact that the cost of string-based methodology is abominably less aside from the code having identifier transforms, line split, removal of enclosure, type, and so forth can't be dissected and recognized whether it is a cloned code or not

### 3.4.2 Token-Based Method

The lexical-based method also known as token-based method. Kamiya et al.,[59] proposed that by lexical analysis the entire source code is partitioned into token and further these token are formed. In this approach each line is divided into tokens by lexer, and forms a single token sequence and generate suffix tree by the set of token string. Further, these token strings are skimmed for detecting copied code. “CCFinder” is one of the token-based primary tools. It used matching algorithm to detect identical sub-sequence of the token string. Baker [50, 51] proposed a tool named as “Dup” which is also based on token-based method. Dup used suffix tree matching algorithm for comparisons and it is also used lexer for tokenization. One of the token-based tool is introduced namely as “CP-Miner” [60, 61] to overcome the hitch of “CCFinder” and “Dup” by using frequent subsequence mining for clone detection rather than sequential analysis. Juergens et at, [62] developed a plug-in in visual studio-based technique. The proposed approach able to identify code clones in Java and C# but unable to handle defects

from programmer side itself. Although, Kawaguchi et al., [63] anticipated the similar method for C ++ and C# but it could not trounce the issue as in [62].

### 3.4.3 An Approach-Based on Syntactic / Tree

In tree-based technique, the source code is classified as dynamic “Abstract Syntax tree (AST)” instead of creating tokens for each announcement and with the assistance of algorithm of tree practically equivalent to sub-tree is investigated in the comparative tree. Baxter et al., [64] has proposed “CloneDR” tool based on AST. AST generate with the help of compiler generator and afterword compare it subtrees using hash function based matrix. In spite of the fact that, it was not ready to recognize indistinguishable clones. Subsequently, Bauhaus has used hash function and matrix of similarities and given a tool name “ccdiml” [65] to overcome the previous problem. Nonetheless, it was awkward to check the renamed identifiers. In sequence, Yang[66] proposed one method, which is based on grammar, utilized for finding the syntactic varieties between the two version of a similar program by making their parse tree and after that applies dynamic programming strategy for distinguishing comparable sub tree. Wahler et al., [67] investigated the way to deal with identify the correct and parameterized clone. Further, Baker first adapts the AST into XML and then used frequent item set data mining method [68] for extorting the clones. Evas et al., [69] gave a further reflection of this methodology by finding near-miss clones and in addition correct clones by utilizing just AST leaves instead of entire AST. Despite the fact that. it couldn't recognize a significant part of the correct clones. Duala Ekoko et al. [70] proposed a tool in Java name “Clone Tracker”. However, due to number of false positive, it was not able to do work for post programming.

Nguyen [71] also proposed tool in java for a clone management, which enhance the time for detection of clone. In spite of the fact that, the previously mentioned overview portrays that gapped clones couldn't be produced by the "AST" and it couldn't see clones if the statements are reordered.

### 3.4.4 An Approach- Based on Syntactic/Metric

Metric-based approach consists of different forms of metrics of the source code fractions named as number of lines, functions etc. Further, these metrics are evaluated in comparison of evaluation of entire source code promptly. The metrics are calculated from the code structure, expression, and control flow for every function features of a code by the Mayrand et al. [17] and after that identical metrics will be retort as clone. Although, few of the metrics are not detected

because of they employed intermediate representation language (IRL) for illustrating every function of the code. IRL unable to detecting segment-based copy-paste which arises frequently while it can identified function-based copy-paste. Kontogiannis et al., [18] render a Markov model based tool named as abstract pattern tool for detecting feasible matches. Further, the clone is identified by metrics and these metrics are extracted form an AST of the code. In addition to this, dynamic programming used for match detection. Though, it can computes the analogy within the code but inept for the exposing of copy-pasted code. Di Lucca. et al. [78] employed an identical methodology for accomplishment the analogy within the static HTML pages by assessing their degree of analogy, which is carried out by estimating the Levenshtein distance of the code [79]. Calefato Lanuible [80, 81] used eMetrics tools for exposing of functional replica and after that identified clones are clustered as per the refactoring opportunities. Further, typically mined code restrained for detecting that is true positive or not. Nevertheless, it could not be accomplished on large scale system. Thus, authors through that metric-based method can detect modest clones from the program.

### 3.4.5 An Approach- Based on Semantic/PDG

AST related issue exercitation by Program dependency graph (PDG) [72-74]. PDG consists of data flow and control flow [75] for semantically and syntactically clone detection. Komondoor and Horwitz [72, 76] projected one of the most prominent approach named as PDG-DUP. it detects PDG subgraph using program slicing technique without changing its semantic behavior. Moreover, Gallagher et al., [77] accomplished the similar slicing-based clone analysis approach. The program slices enumerated by the authors [77] on the entire variables of a program but they are not able to determine any research consequence. An iterative approach within PDG for perceiving maximum sub-graph anticipated by Krinke [73] but it can't employ on a framework to locate the clone. However, PDG-based method can detect non-contiguous clones it is reported by several researchers who are practicing PDG-based approach. Further, they accomplished that PDG can't be used on a huge codebase for clone detection and it will also imply return more false positives.

### 3.4.6 Hybrid / Semantic-Based Method

Hybrid methods consist of two or three methods can be assorted on the on the basis of earlier approaches. Although, many hybrid approach proposed in the literature. Koschke et al., [52] proposed token and tree-based hybrid technique for identifying type-I (exact) and type-II

software clones. This approach creates suffix tree for measuring the AST nodes which is chronological in preorder traversal and after that with the help of suffix tree based algorithm analogies is carried out on the lexical of the AST nodes in place of AST nodes. The function level clones detected by the Microsoft's new Phoenix framework (82], using similar approach. This approach also detects parameterized clones, exact clones with renaming identifier no modifications in data types. Greenan [83] proposed similar approach with the string matching algorithm for the identification of method level clones. AST explored in Euclidean space by Jiang et al., [84] for estimating the vectors and then troop these vectors using Locative Sensitive Hashing (LSH) [85]. Balazinska [86] presented a dynamic pattern matching and classification-based hybrid method. In this approach, method of each body of the method are measured with trait metric and then assessed detected clusters using Patenaude's [87] metric-based method. DeWit [88] proposed a Java language based novel method using dynamic change tracking and resolution. Moreover, it was inadequate for the detection of data flow however; it can detects the replicated code at the developer's level. Additionally, various further techniques for detection in supplementary perspective have been planned (89-94]. At last we can concludes that among the techniques essentially accentuated on the various types of clone detection as type-1 to type-3. Though, However, the aforesaid extensive review on clone detection techniques have been bestowed pictorially in Figure. 3.3.

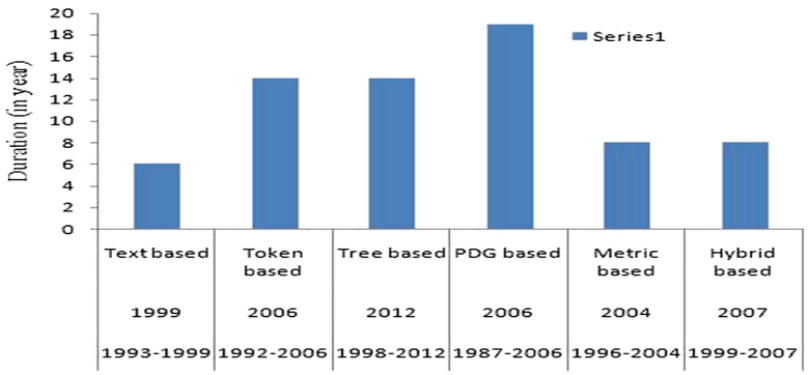



**Figure 3.3:** Comprehensive survey of code clone detection techniques.

## 3.5 OPEN RESEARCH ISSUES IN CODE CLONE DETECTION

Previous survey showed that a number of clone detection techniques have been anticipated [111]. Furthermore, none of the existent approach subsist in the reported survey for the exposing of non-trivial clone detection with a high accuracy, portability, scalability, and

robustness. However, its thorny to describe which tool or method is pragmatic for clone detection due to the their constraints. The textually-identical clone as type1 to type-3 can be facilely detected in comparisons of functionally identical clones as type-4. One of the pioneered approaches named as PDG used for the detection of type-3 to type-4 clones but the main restraint of this method is that it creates various faulty variations of clones, hereby fetching more time for processing a code. Therefore, it is vital to develop an approach which can overwhelm the constraints of existing clone detection method or tools.

## 3.6 CONCLUSION

The software's comprehensibility and maintainability increases due to code replication which is an emerging issue in software development. Thus, software system's quality, structure, and design can be improved by analysis and detection.

This chapter presents a comprehensive review on code clone in perspective of attributes based clone categorization, classification of clone detection tools. Further, it also discusses the detection methods named as "string-based, lexical-based, PDG-based, Metric-based, and hybrid approach by their dimension and sub-dimensions. Although, several clone detection methods have been proposed in the literature [111] but till now efficiency and accuracy is a latent issue in this research era. In addition to this, some of the exist algorithms of clone detection are unable to detect clones from large code base while some of the able to detect only meticulous kind of clones.

This chapter illustrates a general assessment of tools and techniques and research issues in clone detection so that one can simply opt for a suitable system according to the prerequisite, and can study opportunities for "hybridizing" several approaches that may trounce the presented explore issues in clone detection algorithms.

.

# CHAPTER 4

# A NOVEL SOFTWARE PROTECTION APPROACH FOR CODE OBFUSCATION TO ENHANCE SOFTWARE SECURITY

---

The primary objective of this chapter is to propose an approach for securing obfuscation from one of the prominent attack namely as reverse-engineering assaults which are classified as static investigation, dynamic examination, and code clone. To provide software protection against malicious reverse engineering attacks of the software code, there is a need for a peculiar code transformation (semantic-preserving transmutation of a program) method for code protection. Therefore, the objective is to secure a program P by using obfuscation as an outline which impedes the perceptibility of the program text and generates a new program (P'). The transformed code is functionally identical to an "original program" [14, 16], the obfuscation's security is outlined regarding three "reverse engineering attacks as 1) Static analysis" (recapitulate information about code by analyzing). 2) Dynamic analysis attacks (to execute code on different inputs and evaluate their outputs). 3) Code clone attacks (to identify duplicated code from the original program to comprehend the functionality of the software. The foremost ambiguity in forming an adequate "code obfuscator" is to assurance of the security; i.e.to proves that no method can violate code security in real time. Although, several code transformation tools and methods which have been proposed confide on the congenial conception of security and thus can't be measured as provably secure. Various code developers' apprehension is to extend a pioneer and enhanced code transformation method over copyright protection. However, It has been addressed by most of the scientists, and programmers [95-97] that conclusive obfuscation, in general, is imprudent.

## 4.1 INTRODUCTION

In computer security, software piracy is an emerging issue due to its illegitimate alterations. However, the protection of the computer system has been enhanced by substantial endeavors. "Software security is an emerging issue "IT industry" due to it's infringe lead to a significant economical losses. Various protection assaults as "piracy, reverse engineering, and tampering" exploit partly secured code. Further, numerous methods have been suggested for code security from several threats.

Thus, there is a need to evolve a method which secures code from risk analysis, and illegal adaptations. Software security" is an idea of software engineering in this way it keeps code functioning effectively even under malware assaults. Moreover, another significant aspect of software protection is stenography, a branch of cryptography which explores how to transfer data stealthily. The classifications of software security errors are of two types as copyright protection of software code and coding errors in source code. Code protection is a two-player diversion between two assailants: A (programmer) whose essential aim is to hold source code shielded from assault, and B (adversary) who investigates the code and adjust it into a frame which is simple to understand the functionality of code. It is not necessary that the source code of A and B will be the same. However, the code should be reverse engineered in such a way that it should be comprehensible to B, but it is not always necessary that "A" will be able to secure entire code from "B" [7]. According to "Collberg and Thomborson" [4] the copyright protection assaults can be classified as 1) Tampering (unauthorized alterations/ modifications), 2) software piracy (illegal allocation, duplicate and resale of code without authorized privileges), and 3) reverse engineering (evaluating code). Watermarking, obfuscation and tamper- proofing are some of the protection methods which can be used against these attacks. The program gets compact without malfunctioning for an adversary by using code transformation. Code clone attacks, dynamic analysis, and static analysis are the classifications of obfuscation based attacks [9]. The code obfuscation transforms a program P to a new program T (P) in such a way which obstructs the understandability of the code for an adversary without changing its semantic behavior. The transformed code T (P) is functionally identical as original program P [34, 95, 98, 99]. There are two types of code transformations as 1) static transformation (in which transformed code residue to persevere at runtime), and 2) dynamic transformation (to transform a code steadily at runtime, keeping them in constant change which means to obstructs dynamic investigation (5, 100]. The primary objective of this chapter is to propose a peculiar code transformation approach for code protection and an extensive survey on code transformation". The technical contribution of the chapter is illustrated as follows.

- Securing obfuscation against static analysis using semantic clones.
- Securing Obfuscation against dynamic analysis with the help of semantic clones.
- Results are computed with an existing approach using non-trivial software clones through open source tools named as "Dan's tool, Daft logic and Packers tools" and an empirical study is done over obfuscation attacks to prove the legitimacy of the work.

The rest of the section of this chapter is organized as follows.

The basic notions of code transformation, their various form, and aspects are illustrated in Section-2. In addition to this, several types of reverse-engineering assaults and their protection methods are explored in Section-3. Obfuscation's classification depicted in Section-4. Section-5 is entails the thorough review with acknowledged limitations about the research era. Section-6 entails the anticipated approach and their implementation. Section-7 thrashes out the result discussion. Finally, the Section-8 summarizes the chapter.

## 4.2 RELATED WORK

The primary objective of code transformation to make "code's logic" obstruct for an attacker, i.e., to transform a code which is semantically equivalent as the "original code", which is intricate for an adversary to apprehend the features of an application. Code transformation makes code analysis infeasible for an attacker it does not ensure completes protection. In summarize way we can say that it make code analysis compact, while not absurd. Thus, the primary apprehension of most of the software developer is to evolve a novel and enhanced obfuscation technique over intellectual property. Many researchers, scientists, and software developers have been reported in the literature that provable obfuscation, in general, is impractical [34, 95, 96, 98, 99]. "Barak et al., 2012 [34] gave a theoretical investigation of obfuscation. The authors [34] primarily targets on "black-box obfuscation". In black-box obfuscation, obfuscator is envisaged as the compiler that given any information code, yields a code with the comparable usefulness which is entangled for an adversary to secure its facet. However, there are a few code transformation techniques and apparatuses are introduced in the reported works which is employed for transforming a program [97]. Here, few of the transformation approaches, and tools are considered. A "non-trivial code-clone" based transformation approach for software security proposed by Kulkarni and Metta [9]. The proposed method reduces execution speed and enhances the cost. The primary approach of obscurity, for example, identifier renaming which enhances attackers endeavor to understand a given program probed by Ceccato et al., 2009 [100]. Low, 1998 [101] involved a control flow based transformation method for code protection. The "control flow transformations" mainly employed for thrashing the code's logic by using fake control flow. One of the pioneered approach which is proposed by Collberg et al., 1998[102], they presented "expressions, opaque predicates, whose esteem is notable to an obfuscator. Although the significant limitations of this approach it was unable to infer the result of a spontaneous deobfuscator. Wang et al., 2001 [103] present the more a more intricate "control flow flattening" strategy. Moreover, the

anticipated methodology can obstruct a static exploration while Madou et al., 2005 [104] shows that it is inept against hybrid(static and dynamic) assails. The other obfuscation method which is proposed by Collberg et al., [7]. Kulkarni et al., 2104 [9] in which the original source code was improved by code clones. The code segments which are semantically equivalent are compact for an adversary to identify these codes it is depicted by Cohen 1987 [5]. Nonetheless, none of those above techniques and tools approves an annex of techniques based on “code clones and methods which avert against static and dynamic analysis attacks. Moreover, the major problem in existing techniques is execution overhead. The execution overhead initiates a superfluous value for each transformation with respect to execution time and memory utilization intrinsic to execute the obfuscated code and no long haul protection. In this way, the main objective of the proposed methodology is to impede reverse engineering attack and to secure code without variances of the basic function of the code. Thusly, it is requisite to present an adequate code transformation methodology which would overcome the limitations of the current techniques.

## 4.3 PROPOSED METHODOLOGY

The anticipated approach consists of four-steps for securing the “logical part” of the source code. Usually, most of the software applications comprise of logical elements. Thus, in this chapter, we have proposed an approach which can protect source code. At first, to single out logical code segments from the program. Second, to convert it into type-4 code clones or non-trivial clone (syntactically distinct but semantically equivalent) “for each of the logical code fragments. In the third step, to substitute codebase with the replicated code of logical code portions. At last in the fourth step, to exert obfuscation approach on the intact source code to augment the attacker's endeavor to comprehend the transformed code.”

### 4.3.1 Logical Source Code Fragments

In the primary step, we distinguish the legitimate code segments from the codebase. We illustrate a legitimate source code portion to be a code section that accomplishes meticulous value. Each logical sections of the code are either a basic block or a strategy. Besides, in the wake of distinguishing such logical code parts apply following strides on every one of them.

### 4.3.2 Creation of Software Clone

In this step, we generate a functionally identical" (type-4) software clone of a source code fragment F named as code piece F' which has indistinguishable utility as original code fragment

F. Furthermore, to replace F with F' in source code C will make a new Code C' which, for all the data input produce indistinguishable result from P. The software cloning is a method which is used to enhance to reverse engineering efforts stated by Collberg et al., 1997 [7]. The software clones have been asserted on behalf of their similarity as 1) two code segment can be similar by text substance and 2) it can be structurally distinct but semantically similar (functionally identical). Further, the syntactically similar clones are of three types as well as semantically equivalent code clones are assorted in type-4. The intricacy of obfuscated code is amplified due to the inception of software clones for code obfuscation. Although, there are several clone detection methods have been proposed in the literature [20-22] in which one of the tree-based approach namely as "AST (abstract syntax tree)". The AST-based tool named as "CloneDR" was introduced by Baxter et al., 1997 [64] for the detection of textual similarity-based code clone. The tree-based approach diminishes the reverse engineering efforts by detecting and eliminating duplicated code from the source code. The previous literature [20-22] reveals that exist methods able to identify syntactically equivalent code clones from the source code, but unable to identify textually different clones. Figure 4.1 shows the paradigm of "type-1 to type-3 code clones" as Swap 1, Swap2, Swap3 and Table 4.1 depicts the type-4 software clone with the original code fragment. Figure 4.1 represents type-1(similar code fragments except for some alterations in white spaces, comments, newline spaces etc.) type-2 (structurally equivalent code, excluding some alterations as renaming identifiers, data types, variables), and type- 3 (identical copied code with or without further variety; proclamations were included, changed or evacuated).

Table 4.1 demonstrates a paradigm of type-4 (semantically equivalent but syntactically distinct) replica based on identical performance. In this stride, software clones generated by developer manually as depicted in Table 1. Semantically equivalent software clone detection is compact by utilizing any existent static investigation approaches except "program dependency graph (PDG)". The program dependency graph is an amalgam of two or more methodology named as the "control flow graph" and "data flow graph". Generally, PDG is a graphical portrayal of a code. The limitation of PDG is it generates numerous variations of same clone (250) set in the program text of 11,060 lines of code. Furthermore, its performance time isn't comparable to it secures one hour and thirty four minutes to executing a program of 11,060 lines of code.

**Table 4.1:** An Example of semantic software clone

| | |
|---|---|
| 1 /* *original code*/ | 1 /*clone code */ |
| 2 i n t | 2 i n t |
| 3 Swap 1 (i n t a, i n t b) | 3 Swap 2 (i n t a, i n t b) |
| 4 { | 4 { |
| 5 i n t c ; | 5 i n t c ; |
| 6 c=a; | 6 c = a ^ b; |
| 7 a = b; | 7 a = c ^ a; |
| 8 b= c; | 8 b = c ^ b; |
| 9 } | 9 } |

Swap: Int a, b, c; c = a; a = b; b = c; original code

Swap1: Int a, b, c; c = a; a = b; b = c; Type-1

Swap2: Int x, y, z; z = x; x = y; y = z; Type-2

**Figure 4.1:** An illustration of types of software clone

### 4.3.3 Substitution of Original Code with the Replicated Code

Copied code or clone code substitute the original code in this step. The Table 4.1 depicted two java programs for swapping two numbers. At first, original code used temp variable for exchanging while clone code or duplicated code generated using bitwise xor function for swapping two numbers without using the temp variable. Thus, both codes have different approaches for swapping two numbers, but their outcomes will remain the same. Moreover, the functionally identical code clone augments the confrontation of static analysis assaults.

### 4.3.4 Apply Code Transformation on Replicated Code

In this step, the logical part of source code is substituted with the duplicated code or cloned code and then obfuscation technique applied on the entire code. The main aim of the semantic code clone to impede static analysis on source code and enhance the reverse engineering efforts for an adversary. Additionally, one of the other leverage of semantically identical code clones it diminishes the cots and enhances the performance rate in contrast of presented methods.

## 4.4 RESULT AND DISCUSSIONS

The simulation of the proposed approach is done over an “open source java script obfuscator

tools such as Dan's tool [107], Daft logic [108], and Packers tools [109] for executing original source code and cloned code by Java language for transformation. In addition to this, the proposed approach also used an open source java project named as “Gantt project system” for implementation. The evaluation of the anticipated method has been done using “java script obfuscator” tools with the input of the original java source code and clone java code as depicted in Table 4.2.

Table 4.2 demonstrates examples of two java programs for swapping two integer values using distinct methods. In the first approach, the original code used temp variable for swapping two numbers as shown in Table 4.2. The second method in which copied code used “bitwise xor” for transaction two numbers. Both codes have implemented in Java language and they are structurally diverse but functionally similar.

**Table 4.2:** A paradigm of source code and copied code

```
import java.io.*; // Original Code
public class Memoryswap {
  public static void main(String[] args) {
     int a;
     int b;
     int c;
     c= a;
     a=b;
     b=c;
     System.out.println(“ a = ” +a+” b = “+b);
     }
}
```

```
import java.io.*; // Code Clone
public class Xorswap {
public static void main(String[] args)
{
    int a;
    int b;
    int c;
    c=a ^ b;
    a=c ^ a;
    b=c ^ b;
    System.out.println(“a = “+a+” b = “+b);
}
}
```

The results are enumerated with the help of three completely distinct open source “JavaScript obfuscator tools” named as “dan's tool” within the “obfuscator tools named as dan's tool within the year 1990, daft logic within the year 2009 and packers within the year 2009. The two java programs that are as inputs enforced on the eclipse mars tool [110] as illustrated in Table 2. These two programs are taken as input for every JavaScript obfuscator tool dan's tool within the year 1990; daft logic within the year 2009; packers within the year 2009. The anticipated method consists of four steps. Foremost to distinguish the rationale of the source code and after create copied code or clone of the legitimate piece of the source code which is depicted in Table 4.2 As shown in Table 4.2 original source code swapped two numbers using the third variable (temp) and the clone code generated using bitwise xor function for swapping two numbers. The second phase substitute original code with duplicated code or generated clone code. In another step to substitute the original code from the clone code as illustrated in Table 4.2. Thus, the syntactical structure of both codes has been altered without changing their semantic behaviors and then transformed the whole program as shown in Figure 4.5, 4. 6, and 4. 7. Figure 4.5 result

is computed using JavaScript obfuscator for obfuscating the code. In 1990 a JavaScript Obfuscator tool developed named as obfuscator dan's tool. Dan's based on data obfuscation(renaming identifiers), and layout obfuscation approach as eliminate comments from the code for code transformation as shown in Figure 4. 2.

The static analyses become compact for an adversary due to semantic code clone because both codes have similar functionality but the structure is different. Further, the execution speed and reverse engineering efforts will be increased due to the obfuscated code. The outcome of java clone code which is demonstrated in Figure 4.3 is computed from another java script tool named as daft logic which is developed in the year 2009. Daft logic based on layout and data obfuscation for transformation a program.


**Figure 4.2:** An output of obfuscated java clone code using java script obfuscator dan's tool

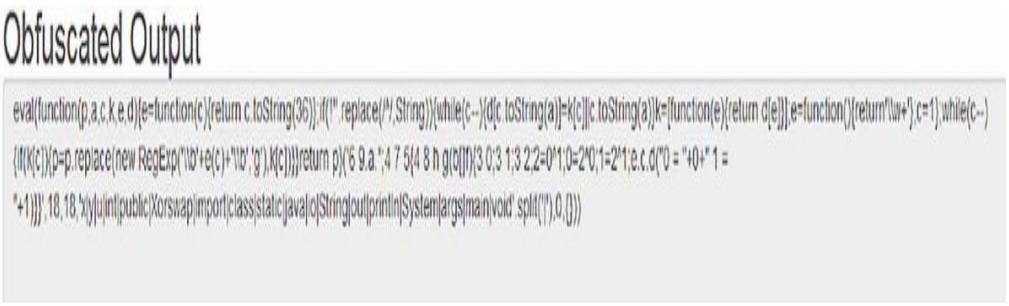



**Figure 4.3:** Obfuscated java clone code output using daft logic tool

Java script obfuscator is an online obfuscator tool which is used for obfuscating a program and make compact for an adversary to comprehend the functionality of code as their outcomes shown in Figure 4.4. The code transformation able to diminish execution time due to its compression quality and it also provides significant protection


**Figure 4.4:** An illustration of obfuscated java code using packer's tool

Figure 4.6 is the outcome of Figure 4.5 which is retrieved by using the online java script obfuscator tool. An open source java project Gantt project system used for this execution. The key motivation for using an open source Gantt project source code (http://www.ganttproject.bizi) is that this code is in millions of lines and by using these codes software clone will be generated of logical part of source code. Further, to reinstate original logical code with the duplicated code clones and then transformed entire code" using data and layout obfuscation. Figure 4.8 shown the obfuscated code which is obscured for an attacker.

Figure 4.6 shows an illustration of java semantic code clones. The source code for implantation is retrieved from an open source java project namely as "Gantt project source code (http://www.ganttproject.biz)", and then semantic software clones which are generated from the logical fragments of source code inserted in the original source code which is shown in Figure 4.6. Comparisions of the proposed approach with the existing method is presented in Table 4.3. The simulation parameters of both approaches have been taken similarly to the same file size as shown in Table 4.3 and Figure 4.7.

Moreover. the proposed approach used an open source java project namely as Gann Project (ganuproject.biz/) and "Ant (ant.apache. org/) systems (ganttproject.biz/)" for implementation. The result comparisons of both techniques are in Figure 4.7.

```
package org.xbill.DNS;

import  java.io.IOException;

public class NameTest extends TestCase
{
    public static class Test_String_init extends TestCase
    {
    private final String    m_abs = "WWW.DnsJava.org.";
    private Name        m_abs_origin;
    private final String    m_rel = "WWW.DnsJava";
    private Name        m_rel_origin;

    protected void setUp() throws TextParseException
    {
        m_abs_origin = Name.fromString("Orig.");
        m_rel_origin = Name.fromString("Orig");
    }

    public void test_ctor_empty()
    {
        try {
        new Name("");
        fail("TextParseException not thrown");
        }
        catch(TextParseException e ){}
    }

    public void test_ctor_at_null_origin() throws TextParseException
    {
        Name n = new Name("@");
        assertFalse(n.isAbsolute());
```

**Figure 4.5:** A paradigm of java semantic code clone

Obfuscated Output

```
eval(function(p,a,c,k,e,d){e=function(c){return(c<a?'':e(parseInt(c/a)))+((c=c%a)>35?String.fromCharCode(c+29):c.toString(36))};if(!''.replace(/^/,String)){while(c--){d[e(c)]=k[c]||e(c)}k=[function(e)
{return d[e]}];e=function(){return'\\w+'};c=1};while(c--){if(k[c]){p=p.replace(new RegExp('\\b'+e(c)+'\\b','g'),k[c])}}return p}('3a 1o.2h.2g;1v 2t.32.1W;1v 2t.2X.z;1v 2j.2l.2v;1v 2j.2l.1l;1v 2j.2l.2i;1v
1o.2h.2g.9;1v 1o.2h.2g.kq 1b 2l 1w 1l[q 1u 1b 2E 1w 1l[1X 20 1t 1O="1D.1p.1o.";1X 9 1m;1X 20 1t 1T="1D.1p";1X 9 1x;3E t 3J()u k{1m=9.G("2k.");1x=9.G("2k")}q t 3K(){T{h 9("");R("k S Z")}Q(k
e){}}q t 3P()u k{9 n=h 9("@");L(n.14());L(n.11());j(0,n.X());j(0,n.Y())}q t 3Q()u k{9 n=h 9("@",1m);j(1m,n)}q t 3O()u k{9 n=h 9("@",1x);j(1x,n)}q t 3L()u k{9 n=h
9(".");j(9.1s,n);3s(9.1s,n);j(1,n.X());j(1,n.Y())}q t 3q()u k{9 n=h 9("*");L(n.14());x(n.11());j(1,n.X());j(2,n.Y());x(z.y(h l[]{1,\'*\'},n.C(0)));j("*",n.E(0))}q t 3p()u k{9 n=h
9(1O);x(n.14());L(n.11());j(4,n.X());j(17,n.Y());x(z.y(h l[]{3,\'W\',\'W\',\'W\'},n.C(0)));j("1D",n.E(0));x(z.y(h l[]{7,\'D\',\'n\',\'s\',\'J\',\'a\',\'v\',\'a\'},n.C(1)));j("1p",n.E(1));x(z.y(h l[]
{3,\'o\',\'r\',\'g\'},n.C(2)));j("1o",n.E(2));x(z.y(h l[]{0},n.C(3)));j("",n.E(3))}q t 3o()u k{9 n=h 9(1T);L(n.14());L(n.11());j(2,n.X());j(12,n.Y());x(z.y(h l[]{3,\'W\',\'W\',\'W\'},n.C(0)));j("1D",n.E(0));x(z.y(h l[]
{7,\'D\',\'n\',\'s\',\'J\',\'a\',\'v\',\'a\'},n.C(1)));j("1p",n.E(1))}q t 3t()u k{9 n=h 9("a.b.c.d.e.f.");x(n.14());L(n.11());j(7,n.X());j(13,n.Y());x(z.y(h l[]{1,\'a\'},n.C(0)));j("a",n.E(0));x(z.y(h l[]
{1,\'b\'},n.C(1)));j("b",n.E(1));x(z.y(h l[]{1,\'c\'},n.C(2)));j("c",n.E(2));x(z.y(h l[]{1,\'d\'},n.C(3)));j("d",n.E(3));x(z.y(h l[]{1,\'e\'},n.C(4)));j("e",n.E(4));x(z.y(h l[]{1,\'f\'},n.C(5)));j("f",n.E(5));x(z.y(h l[]
{0},n.C(6)));j("",n.E(6))}q t 3u()u k{9 n=h 9("a.b.c.d.e.f.g.");x(n.14());L(n.11());j(8,n.X());j(15,n.Y());x(z.y(h l[]{1,\'a\'},n.C(0)));j("a",n.E(0));x(z.y(h l[]{1,\'b\'},n.C(1)));j("b",n.E(1));x(z.y(h l[]
```

**Figure 4.6:** An example of obfuscated java code

**Table 4.3:** A comparison of the proposed approach with existing method

| Metric | Exist Approach | Proposed Approach |
|---|---|---|
| 10kb | 1s | 0.00025 |
| 25kb | 2s | 2.642 |
| 42kb | 7s | 5.782 |

The proposed method has been implemented using three online java obfuscators tools, and their outcomes are shown in Figures 4.2, 4.3, and 4.4. The performance of both methods will be compared by two parameters. The first parameter is to compare regarding file size and second parameters in terms of execution time. Figure 4.7 show the execution time of the proposed method is far better in comparisons of an existing approach.

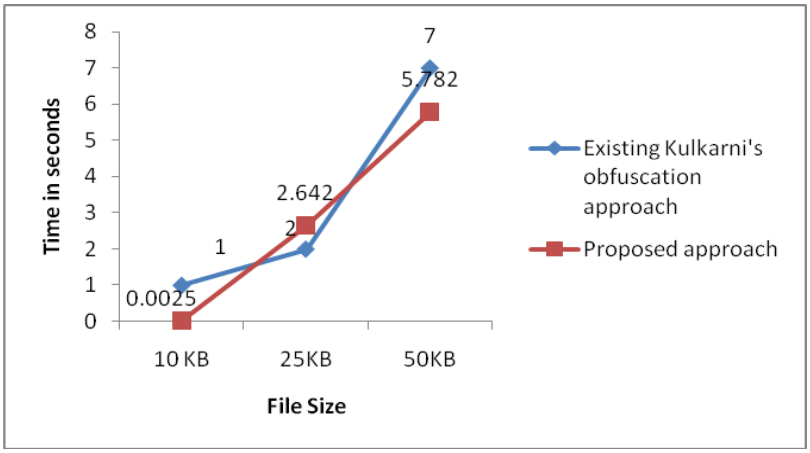



**Figure 4.7:** Comparisons with Existing Kulkarni Obfuscation Approach

## 4.5 CONCLUSION AND FUTURE WORK

This chapter presents a peculiar obfuscation technique for code security. The anticipated method employed an open source java project named as "Gantt project" and open source java obfuscator tools for implementation. Further, the proposed method is compared with the existing method using the same parameters. Though, introduced technique secure code against static analysis and augmented complexity of dynamic analysis for an adversary. In static analysis an adversary analysis structure of source code for comprehending the functionality of source code. Thus, to address this proposed approach at first identified logical segments of source code thoroughly averted. However, the proposed method makes assault considerably thornier so data deduced from one execution cycle of the program by a virtual machine that doesn't necessarily help in detaining the nature of the computer code. Moreover, this runs on different inputs within the same manner. In addition to this, the primary objective of the proposed methodology is to thwart static analysis peculiarly and makes dynamic analysis thwart for an attacker. Moreover, the proposed approach increase execution speed and abates cost as well as accumulates the endeavors of an adversary. Note that the proposed method doesn't shall invent an idyllically protected technique against "dynamic analysis". In future communication, a safer obfuscation approach should be provided against dynamic analysis with performance metrics ("potency, resilience, stealth").

# CHAPTER 5
# DETECTIONOF SOFTWARE CLONES

The objective of this chapter is to propose the approach for the exposing of type-1 to type-3 code clones from the source code with higher precision, recall, portability, scalability, and robustness. Reusing code fragments with some adaption is known as software cloning, and the copied code is called software cloning. The code clones are two types on the basis of their attributes as textual similarity-based ("type-1 to type-3") and functional similarity-based (type-4). Further, many detection approaches have been anticipated in the existing survey [111] for the revealing of "type-1 to type-3" code replica, but each technique has its pros and cons. Thus, it is essential to develop an approach which can detect type-1 to type-3 code clones with higher "precision, recall", portability, scalability, and robustness. This chapter is divided into two sections. The remainder of the chapter is organized as stated below. Section 5.1 depicted the hybrid approach for the detection of type-1 code clone. Section 5.2 presents the proposed approach for type-2 code clone detection by "directed acyclic graph" (DAG).

## 5.1 DETECTION OF TYPE-1 SOFTWARE CLONES USING A HYBRID TECHNIQUE

Over the last decade, numerous detection methods and tools have been developed. Copying a section of source code from one place and paste them into another place with or without some alterations in the code base is a frequent activity is known as code cloning, and the replicated fragments is known as code clone. This section illustrates a competent technique for type-1 code clone detection. The anticipated method detects type-1 code clones with high accuracy namely as recall precision portability, and scalability. Further, mutation operator-based editing nomenclature used for generating the type- 1 code clones.

### 5.1.1 Introduction

The software clones introduced in code base by software developer by copying/pasting activities with or without significant alterations. Software clone is copied fragments of source code which often occurs due to replication in source code [20, 111]. Empirical studies illustrate that the existence of software clones usually reduces the software maintainability and quality of

software [112-113]. Suppose if one code segment S1 contains an error (bugs) and they are customized into another location as S2 code segment. Further, if faults included into Si code segment then S2 code segment must contain the same bug, and they must be corrected at the same time when a flaw in Si is fixed. If the software developer does not identify the presence of a software clone, then it may be the cause of depreciation of software quality. Although, it is often said by several researchers that software cloning is a frequent process in code development and that a significant portion of program text (between 20-59%) is replicated or customized from previously executed code blocks or segments [15, 21, 114, 115]. Thus, it is essential to identify the duplicate portion from the large-scale source code [15, 114]. The software clones are of two types by their attributes as a syntactic attribute and semantic attribute. The syntactic attribute-based replicated codes are three types as type-1 to type-3 while semantic attribute-based software clones are type-4[21]. The description of type-1 to type-4 software clones are described below:

Type-1: Transform the layout of code via altering the whitespaces, comments, blank spaces, etc. without changing the code itself.
Type-2: Modifications in the data types of variables, renaming of variables and functions.
Type-3: Insert or delete a new line and alter the expressions without modifying its behavior.
Types-4: Alter the code structure without changing its functionality.
The obscurity in clone detection will be increased according to the modification level of code. The existing literature reveals that some copied code detection approaches and tools have been developed [116-117]. According to some facets, code clone detection methods can be classified as "string-based, lexical-based, tree-based, metric-based, program dependency graph-based, and hybrid approaches" [116]. Each method has its pros and cons. Presently; several researchers have presented many replicated code detection approaches and tools in the literature [116-121]. In this section, we have introduced a method which can detect type-1 software clones and these clones will be created using an editing taxonomy which is based on mutation-operator.
The potential augmentations in this chapter are mentioned below:

- Type-1 software clones generated using mutation operators.
- Type-1 clone detection from large-scale open source software.
- Compared proposed approach with existing methods.

### 5.1.2 Related Work

The existing literature reveals that a number of clone detection approaches have been anticipated in the existing work [20, 21, 22, 111, 119-121].

### 5.1.3 Proposed Approach

The proposed approach consists of four steps for detection of exact software clones (type-1).The flowchart of the proposed approach is shown in Figure5. 1.

A. **Software clones creation by mutation operators:** In this step, mutation operators used for generating exact clones.

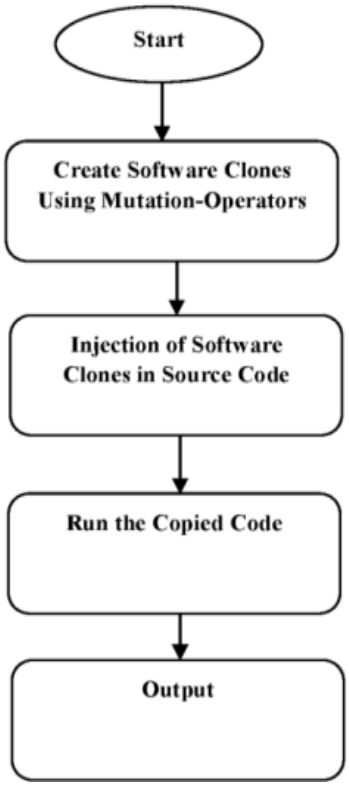



**Figure 5.1:** Flow chart of proposed approach

Figure 5.2 depicts the mutation operators types for generating exact code clones as 1) mRW(removing whitespaces),2)mCNWs(add newline spaces), 3)mVF(change the formatting)4)mMCs(alterations in comments), and 5) mTBs(changes in blank spaces) [121]. These operators are used to generate exact clones types which are shown in Figure 5.3.

Figure 5.3 shows the exact clones(type-1) using mutation operators as Si (a) amendments in white spaces in S1(b) changes in comments in S1(c) making alterations in formatting in S 1(d) insert blank spaces in S 1(e) new line spaces

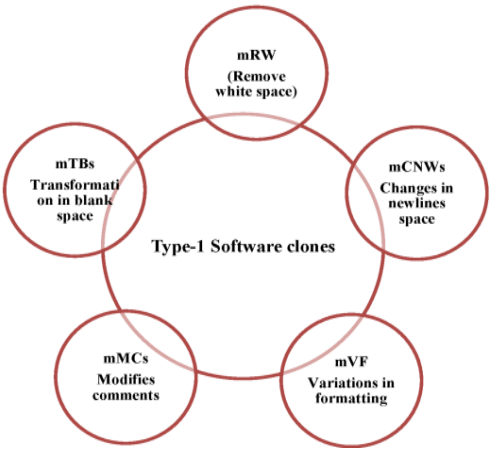



**Figure 5.2:** Exact clones (type-1) using mutation operator

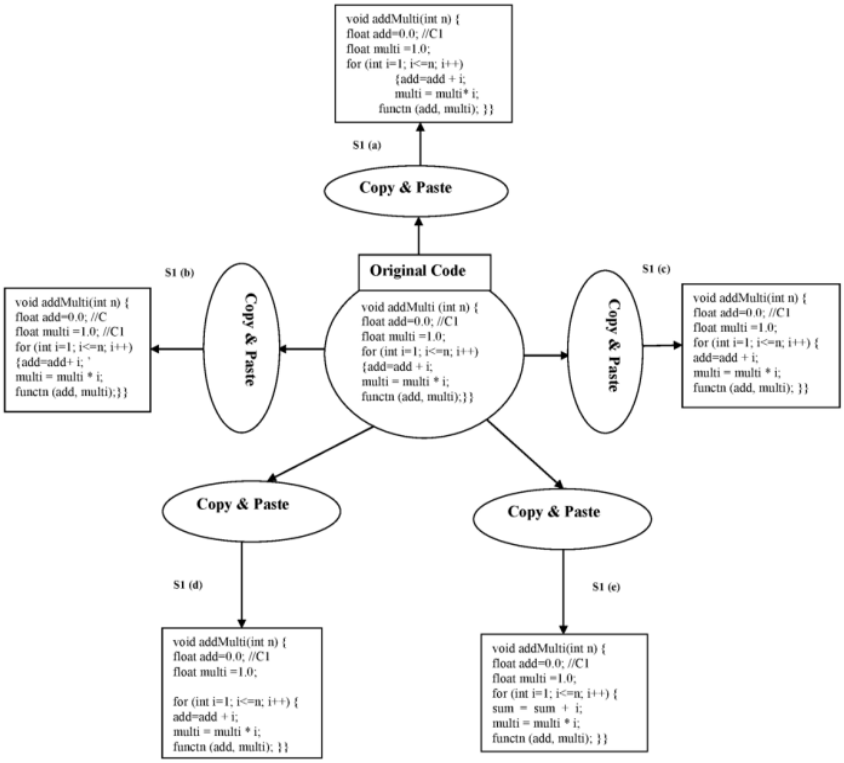



**Figure 5.3:** An illustration of exact clones (type-1) using mutation operator

**B. Insertion of replicated code fragments in codebase:** In this step, software clones injected in the open source project named as wet lab[122] and these software clones are generated by using mutation operator which is shown in Figure 5.2 or 5.3.

**C. Detection of software clones using the proposed approach:** Software clones are detected using the proposed approach.

**D. Comparisons with existing methods:** The proposed approach is compared with existing methods which are described in the result discussion section.

### 5.1.4 Result Analysis

We have used an open source project wet lab(c language) [122] for code clone injection which is generated using mutation operator-based editing taxonomy. Further, the code blocks tool [123] is used for implementation. The Comparisons of the proposed approach with existing methods are shown in Table 5.1 as well as in Figure 5.4 by metrics (recall, precision, portability, scalability, robustness). Further, the implementations have been conducted on the open source project wet lab[122] source code of c language on different files which is present in Table 5.2.

**Table 5.1:** Comparisons of proposed approach with existing methods

| Tool | Precision | Recall | Portability | Scalability | Robustness |
|---|---|---|---|---|---|
| **PMD[124]** | 46 | 59 | Y(highly) | Y(highly) | Y(highly) |
| **CCFinderX [125]** | 56 | 51 | Y(highly) | Y(highly) | X(limited) |
| **CPMiner[126]** | 41 | 48 | Y(highly) | X(limited) | X(limited) |
| **Bauhaus [65, 127]** | 81 | 49 | X(limited) | Y(highly) | X(limited) |
| **Proposed Approach** | 100 | 100 | Y(highly) | Y(highly) | Y(highly) |

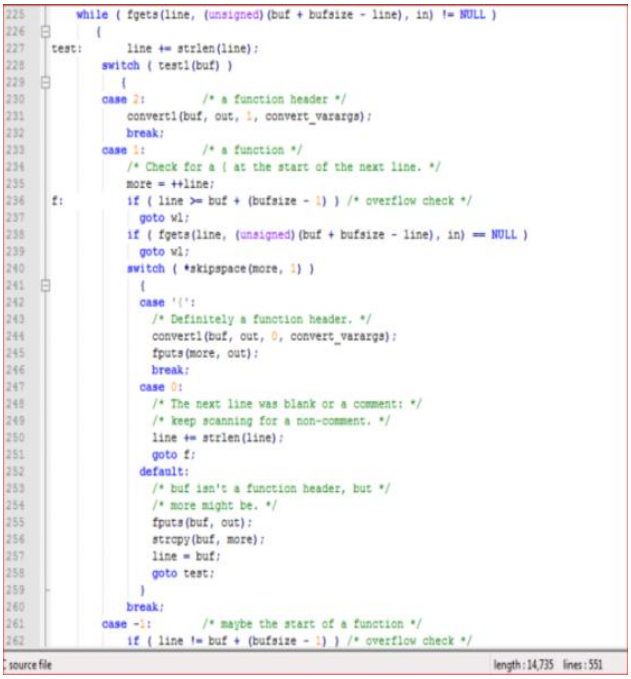
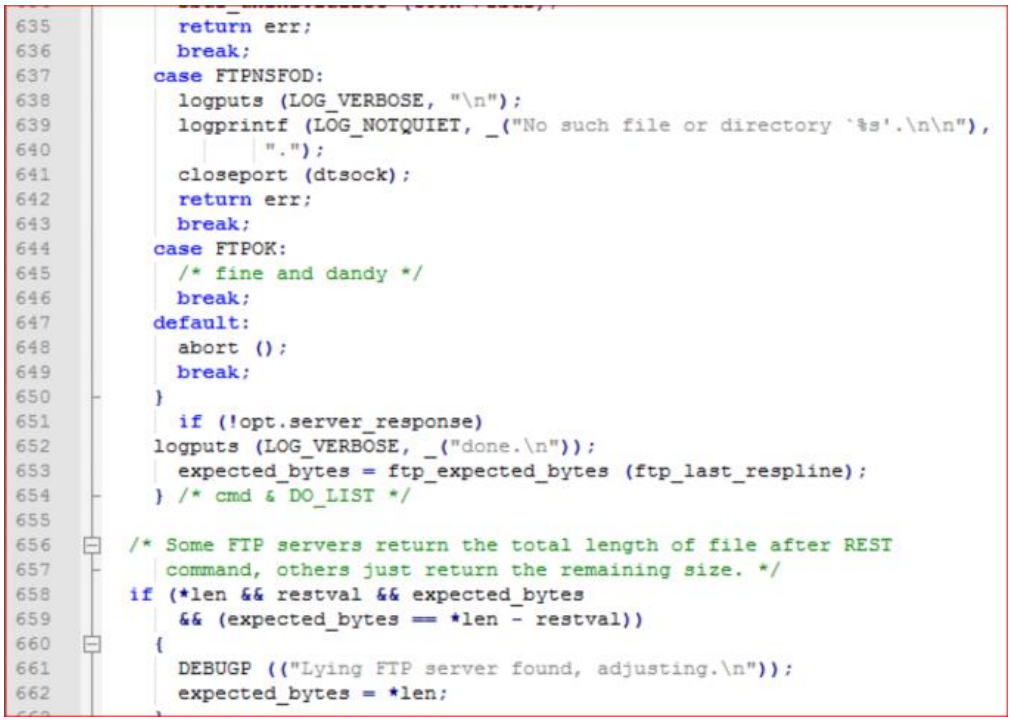



**Figure 5.4:** An example of wet lab open source original code with mRW clone

```
/
*Writeblanksoverpartofastring.
*Don'toverwriteend-of-linecharacters.
*
int
writeblanks(start,end)
char*start;
char*end;
{       char*p;
        for(p=start;p<end;p++)
        if(*p!='\r'&&*p!='\n')
        *p=' ';
        return0;
}
*Testwhetherthestringinbufisafunctiondefinition.
*Thestringmaycontainand
*Returnasfollows:
*       0-definitelynotafunctiondefinition;
*       1-definitelyafunctiondefinition;
*       2-definitelyafunctionprototype(NOTUSED);
*       -1-maybethebeginningofafunctiondefinition,
*               appendanotherlineandlookagain.
*Thereasonwedon'tattempttoconvertfunctionprototypesisthat
*Ghostscript'sdeclaration-generatingmacroslooktoomuchlike
*prototypes,andconfusethealgorithms.
*
int
test1(buf)
char*buf;
{       registerchar*p=buf;
        char*bend;
        char*endfn;
        intcontin;
        if(!isidfirstchar(*p))
        return0;
        bend=skipspace(buf+strlen(buf)-1,-1);
        switch(*bend)
        {
        case';':contin=0
        case')':contin=1;break;
        case'{':return0;
        case'}':return0;
        default:contin=-1;
        }
        while(isidchar(*p))
        p++;
        endfn=p;
        p=skipspace(p,1);
        if(*p++!='(')
        return0;
        p=skipspace(p,1);
        if(*p==')')
        return0;
```

**Figure 5.5:** An illustration of detection of mRW from original source code



```
635       return err;
636       break;
637     case FTPNSFOD:
638       logputs (LOG_VERBOSE, "\n");
639       logprintf (LOG_NOTQUIET, _("No such file or directory `%s'.\n\n"),
640                  ".");
641       closeport (dtsock);
642       return err;
643       break;
644     case FTPOK:
645       /* fine and dandy */
646       break;
647     default:
648       abort ();
649       break;
650     }
651       if (!opt.server_response)
652     logputs (LOG_VERBOSE, _("done.\n"));
653       expected_bytes = ftp_expected_bytes (ftp_last_respline);
654     } /* cmd & DO_LIST */
655
656   /* Some FTP servers return the total length of file after REST
657      command, others just return the remaining size. */
658   if (*len && restval && expected_bytes
659       && (expected_bytes == *len - restval))
660     {
661       DEBUGP (("Lying FTP server found, adjusting.\n"));
662       expected_bytes = *len;
```

**Figure 5.6:** An illustration of wet lab open source original code with mCNW clone

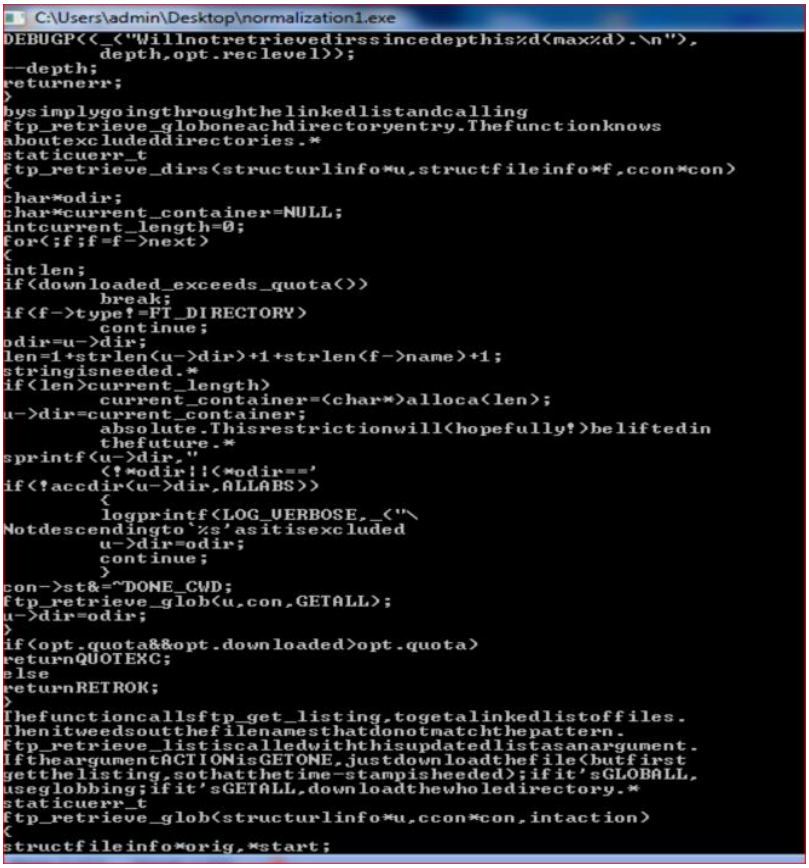



**Figure 5.7:** An illustration of detection of mCNW from original source code

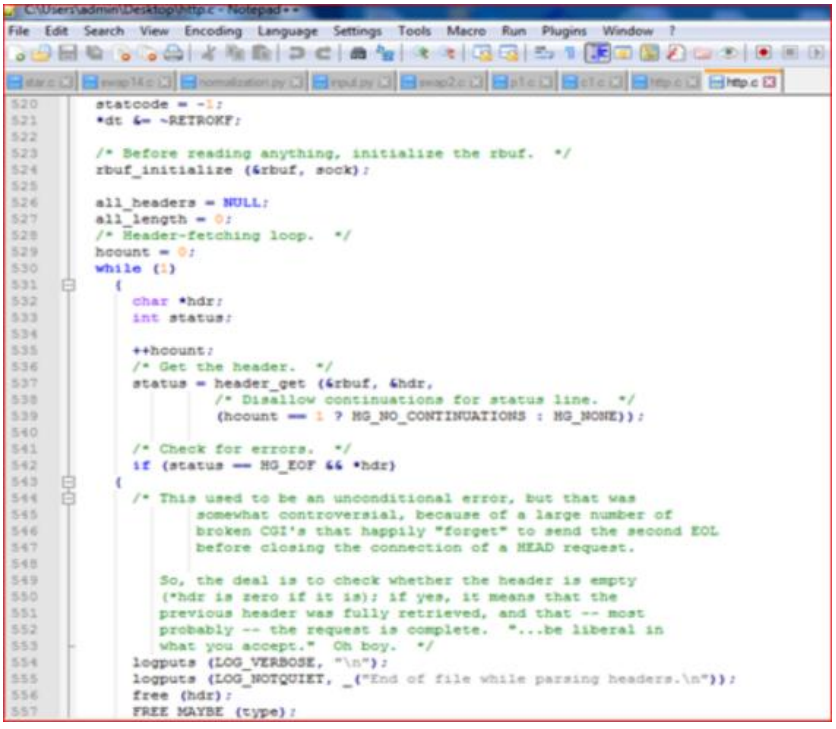



**Figure 5.8:** An illustration of wet lab open source original code with mVF clone

**Figure 5.9:** An illustration of detection of mVF from original source code

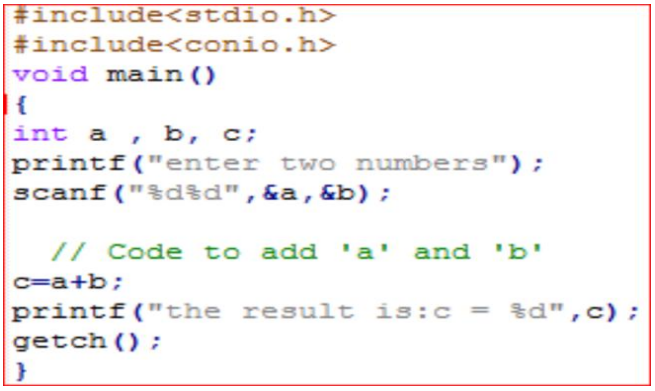


**Figure 5.10:** An illustration of detection of mCs and mTBs from original source code

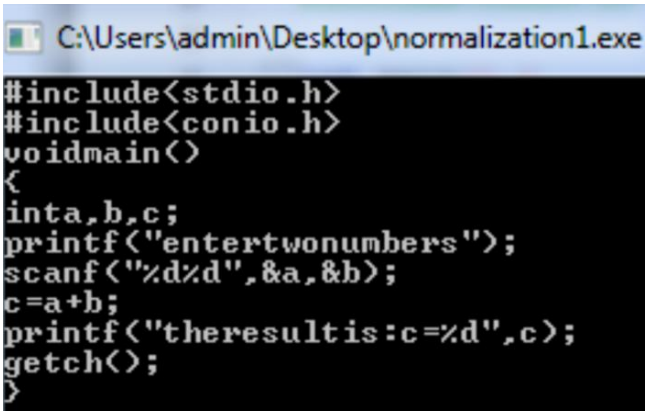


**Figure 5.11:** An example of detection of mCs and mTBs from original source code

**Table 5.2:** Running time of proposed method

| LOC | Proposed Approach | Exist method |
|---|---|---|
| 551 | 0.995s | 2 |
| 1192 | 1.185s | 2.5 |
| 1743 | 2.501s | 78 |

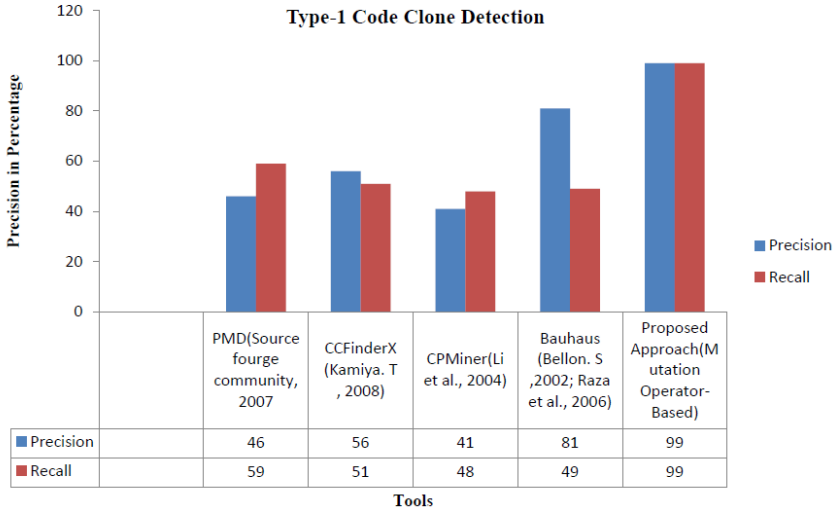



**Figure 5.12:** Comparisons of proposed method with existing methods

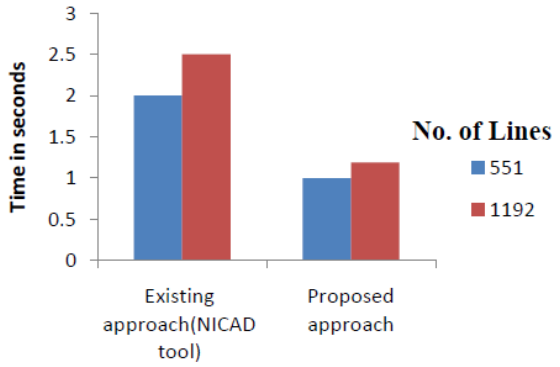



**Figure 5.13:** Running time Comparisons of proposed method with existing method (NICAD tool)

**Table 5.3:** Running time of proposed method and existing method

| File Size | Proposed Approach |
|---|---|
| 15k | 0.995s |
| 26K | 1.185s |
| 42K | 1.80s |
| 51K | 2.501s |

### 5.1.5 Conclusion

The proposed approach can detect exact ("type-1") code replica. Further, the developed method is evaluated with existing methods by effectiveness criteria as scalability, portability, precision, recall, robustness. Although, the running time is also calculated; of the proposed method with

the existing approach by different file sizes and line of code which are shown in Table 5.2 to Table 5.3. In future, we will compare our approach on many different subject systems as java, net beans, etc. with different clone detection tools which are based on different methods.

## 5.2 DETECTION OF TYPE-2 SOFTWARE CLONES USING DAG

Reusing code fragments with significant modifications in the source code is the severe maintenance problem in the software industry. The software clone enhances the efforts when the code is increased. It also reduces software quality due to replication procedure which is a common process in code expansion. Although, several code clone detection approaches have been developed in the past few years and each method has its tunable parameters for software clones detection. Thus, this chapter proposed a compiler optimization technique named as "directed acyclic graph (DAG)" for renamed code clone detection. Although, in compiler optimization the common sub-expressions are eliminated from the source code by DAG. Therefore, proposed approach used DAG for renamed code clone (Identical code fragments as type-1 except for some variations in variable names and function names) detection.

Moreover, the proposed approach is also used mutation operators-based editing taxonomy for generating various types of renamed software clones. The proposed approach is compared with existing methods on some sample source code. Further, the implementation results show the efficiency of the proposed approach with the existing method.

### 5.2.1 Introduction

The software cloning is customary activity of copying and pasting of code fragments from one place to another place with significant modifications in source code. A fault is detected in the section of code then it should be corrected in the entire source code. Software cloning reduces software quality and increases maintenance problem due to copying and pasting process. Thus, software cloning is an emerging issue in the software industry, and it is imperative to propose a method which can detect duplicate parts from the source code. The code clones have four types by their characteristics as syntactic "textually equivalent code clones" and the "functional similarity-based software clones". Syntactically equivalent clones can be classified into three types as type-1(exact copied code with minor changes in white spaces, comments, blank spaces, etc.) type-2(structurally similar clones as type-1 with some modifications in variables names, function names, etc.) type-3(syntactically identical with some insertion of new statements or

deletion of comments etc.). The semantically identical code clones are categorized into types-4, and these types of clones will remain the similar in their functionality but distinct layout [20, 21, 111]. This chapter presents a "directed acyclic graph" for the exposing of renamed code clones or type-2 software clones [128]. Moreover, in this chapter, we have also used mutation operators-based editing taxonomy for generating some type 2 software clones. The potential augmentations in this manuscript are mentioned below:

- Type-2 clone(renamed code clones) detection by DAG.
- To compare two "directed acyclic graph" by using the Kendall-tau method
- By utilizing an editing taxonomy of mutation operators to generate various type of software clones.

### 5.2.2 Related Work

A comprehensive review has been through by several scientists on software clone. The absolute realm of code clone apprehension approaches and tools consists of various detection methods namely as "text-based, token-based, graph-based, metric-based, and hybrid are discussed in the existing literature [20, 21. 22. 111, 129-131].

### 5.2.3 Proposed Approach

This section thrashes out a method known as the "directed acyclic graph" for the exposure of renamed code clones. The proposed has four steps which are shown below in Figure 5.14. At first, mutation operator-based editing nomenclature used for generating renmaed code clones. In second phase, generated code clones will be translated into three address code. Third step generate the "directed acyclic graph" by three address code and finally, the Kendall-tau method used for comparing generated "directed acyclic graph". Directed acyclic graph (DAG) have some properties which are described [128]. The directed acyclic can be defined as mentioned below:

"A directed graph G = (V, E) is a tuple of vertices V and edges E, which is a set of ordered pairs (i, De VW. A DAG has no directed cycles [9]. The steps of the proposed method are described below [128]":"

**A. Using mutation operator generate code clones:**

The code clones generated by mutation operators in this steps which are shown in Figure 5.15.

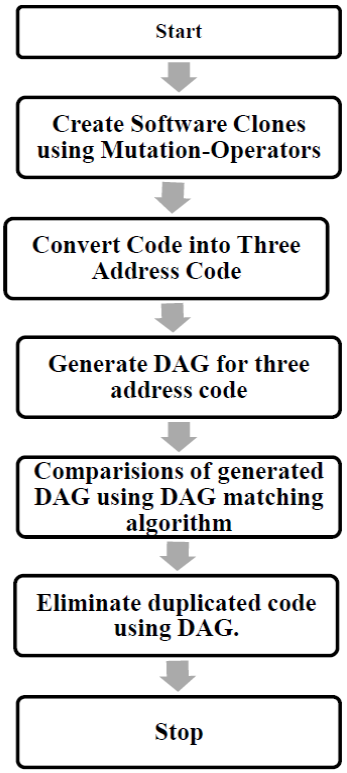



**Figure 5.14:** Flow chart of proposed approach

**B. Transform code into Three Address Code:**

This step convert mutated code which are shown in Figure 5.16 and Table 5.4 and after that it will converted into "three address code" as shown in Table 5.5.

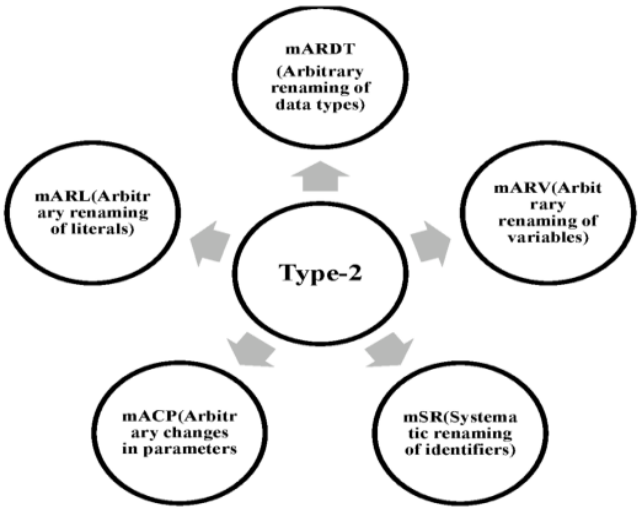



**Figure 5.15:** An illustration of mutation operators

Figure 5.15 shows the types of mutation operators for generating type-2 software clones as 1) mARD (Arbitrary renaming of data types) 2) mARV (Arbitrary renaming of variables) 3) mSR (Systematic renaming of identifiers 4) mACP (Arbitrary changes in parameters 5) mARL (Arbitrary renaming of literals).

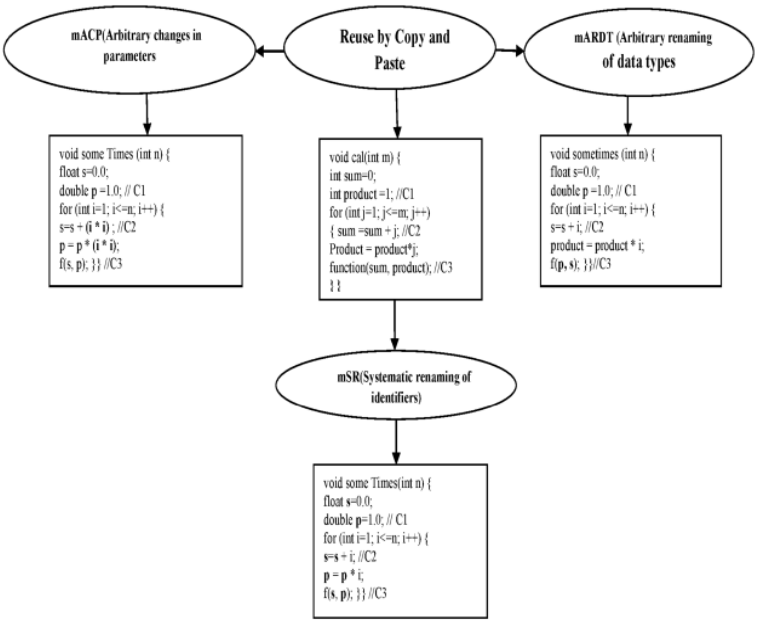



**Figure 5.16:** An illustration of type-2clones using mutation operator-based editing taxonomy

### C. DAG generation for Three Address Code

This step generates DAG after transforming code into three address code which is shown in Table 5.6 [4].

### D. Comparisons of DAG

In this step code, Cl and C2 DAG's will be compared by using the Kendall-tau distance algorithm. The Kendall-tau [9] distance algorithm characterized over pairs of vertices in the two "DAGs". Further, two pair of vertices may not be exactly analogous in DAGs. In two DAGs a pair of vertices may be neither concordant nor discordant. Thus, a distance computes that envisage couple of vertices in two DAGs should identify the below mentioned two cases of assigning a penalty."

**Discordant pairs:** To "penalize an edge (i, j) if I precede j in one graph while j precedes i into another graph. Potentially discordant pairs: if one or both input DAGs are not fully connected then there are a pairs of vertices (i, j) for which we don't know if I precedes j or vice versa, so it may be possibility of potentially discordant pair, which should be penalized less heavily than a pure discordant pair. On the other side of "DAGs" comparisons, one of the most pioneer approaches is to compare two DAGs by an arbitrary ordering of the vertices [129]."

In this approach, two DAGs will be compared regarding "penalty parameters p and q where 0<=p, q<=1. This ordering assumed only for notational simplicity, and it is not related to the

concept of total order. Such kind of ordering only used to guarantee that each pair of vertices I and j are considered only once, so any arbitrary bijection from the set of vertices V to (I M) can be used. However, the definition of DAG distance is based on the ordering of vertices; the outcome does not depend on it."

**Table 5.4:** An Example of original source code and copied code

| Original Code (C1) | Duplicated Code (C2) |
|---|---|
| #include<stdio.h><br>int main()<br>{<br>int x, y, z, n;<br>printf("Enter three numbers to add\n");<br>scanf("%d%d%d", &x, &y, &z);<br>n = (x + y)* (x+ y+ z);<br>printf("Sum of entered numbers = %d\n",n);<br>return 0;<br>} | #include<stdio.h><br>int main()<br>{<br>int a, b, c, t;<br>printf("Enter three numbers to add\n");<br>scanf("%d%d%d", &a, &b, &c);<br>t = (a + b)* (a+ b+ c);<br>printf("Sum of entered numbers = %d\n",t);<br>return 0;<br>} |

**Table 5.5:** Transformation of code into three address code

| Conversion of original code into Three Address Code | Conversion of duplicated code into Three Address Codes |
|---|---|
| s1 = x + y; | t1 = a + b; |
| s2 = s1+ z; | t2 = t1+ c; |
| s3= s1 + s2 | t3= t1+ t2; |

Let G1 = (V, El) and G2 = (V, E2) "be two DAGs and let 0 <= p; q <= 1 be fixed parameters. Let i, and j be two vertices of V such that i < j. Let e = (i; j) and f = (j; i) be e with reversed direction. We define Ki, j(p, q) (G1, G2) to be a penalty associated to vertices i and j for the DAGs G1 and G2 concerning the parameters p and q. We consider four cases:"

Case 1: (e ∈ El and e ∈ E2) or (f ∈ El and f ∈ E2). In this case, GI and G2 agree on e, and we set the distance to be K i, j (G1, G2) = 0.

Case 2: (e ∈ El and f ∈ E2) or (e ∈ E2 and f ∈ El). In this case, G1 and G2 completely disagree on e, and we set the distance to be K i, j (G1, G2) = 1.

Case 3: (e or f ∈ El and e, f ∈ E2) or (e or f ∈ E2 and e, f ∈E1). In this case, "an edge between i and j exists in one graph but not in the other. We set K i, j (Gl; G2) = p. Elimination of C2 (copied code) from Cl (source code). In this step, the copied code is eliminated by using a directed acyclic graph [4]."

**Table 5.6:** Comparisons of DAG

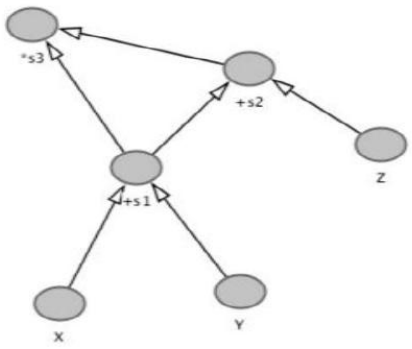
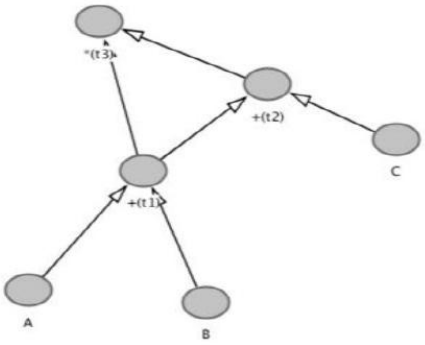
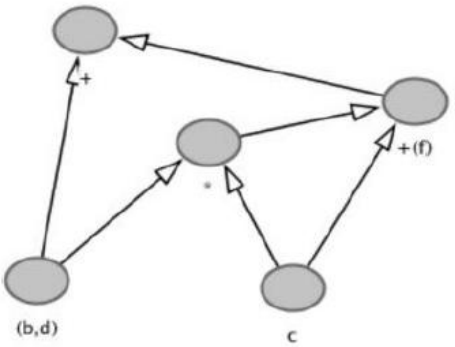
| Original code C1 DAG created by using three address code | Copied code C2 DAG created by using of three address code |
|---|---|
|  |  |





**Table 5.7:** Elimination of copied code

| Eliminate copied code | Corresponding DAG of optimized code |
|---|---|
| 1. a = b * c;<br>2. d = b;<br>3. f = b + c;<br>4. g = f + d; |  |



### 5.2.4 Result Analysis

DAG execution performed by "DAGitty" [130] which is an open source tool also used for generating DAG for the original source and copied code as shown in Table 5.6. Furthermore, both DAGs are compared by the DAG distance measure approach [131]. The main objective of DAG distance measure is to compare two DAG by "arbitrary ordering of the vertices of the two DAGs." The DAGs distance measure can be calculated by using the following equations:

Definition 1: Let G = (V, El) and G2 = (V, E2)"be two DAGs and let 0<=p, q<=1 be fixed parameters. We define the DAG distance K(p; q)(G1 ;G2) to be the sum of distances over pairs of vertices (i; j) ∈ V *V such that i < j, K(p, q)(G1,G2) = Σ//(i; j) ∈V *V Ki,j(p, q) (G1;G2) Whenever clear from the context, we will drop p and q from the notation and write K(G1; G2)" instead of K(p, q) (G1, G2).

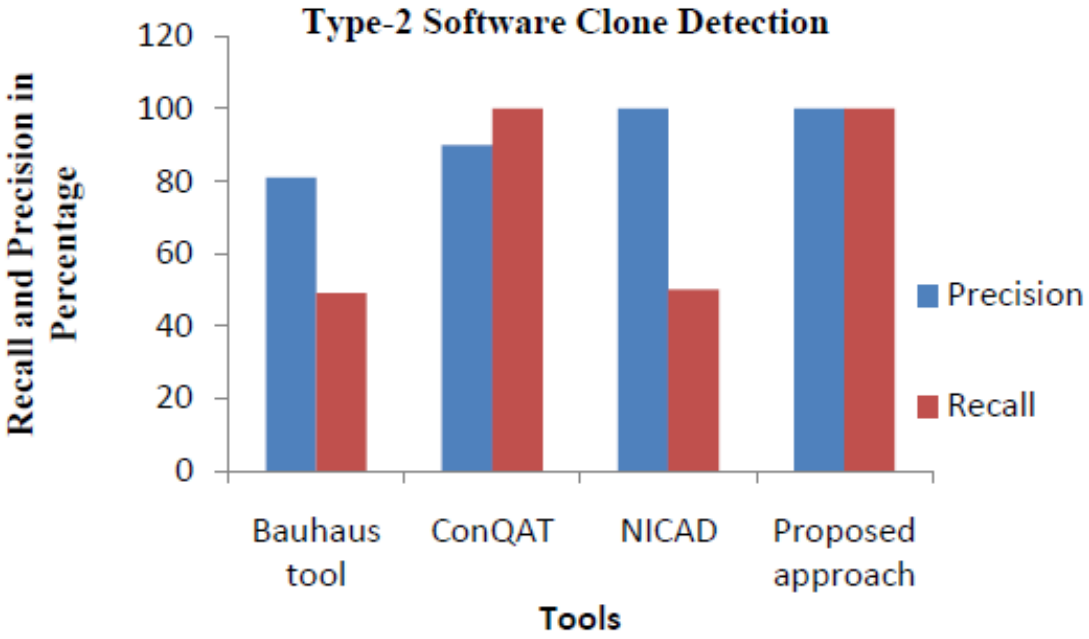



**Figure 5. 17:** Comparison of proposed approach with existing methods

### 5.2.5 Conclusion

This section discussed Directed acyclic graph (DAG) based approach for the detection of type-2 code clones from the code base. Although, DAG is basically used in optimization for eradicating common sub-expressions from the source code. Hence, proposed approach used "DAG" for the exposing of renamed software clones because of renamed clones which is a form of "type-2 code clones" (identical code fragments with few changes in renaming variables, identifiers, function names, etc.). To the best of author's knowledge, DAG is used the first time for the detection of software clones. Moreover, a DAG comparison algorithm is used for comparing generated graphs of two source code as well as mutation operators also used for creating a maximum number of software clones. In the future, we used DAG for various types of open source systems as c, c++, java, net beans, etc.

# CHAPTER 6
# CORRELATION AND PERFORMANCE ESTIMATION OF CLONE DETECTION TOOLS

Over the past few decades many tools and methods have been anticipated by numerous scientists to perceive duplicated automatically in a codebase. Though, it is not mentioned in the literature how to evaluate performance of these software clone detection tools in perspective scalability, precision, recall, and portability. Moreover, each tool has its own merits and demerits but, the user's requirement depends on the tool's application. Thus, it is essential for the user that they should be apprehensive about the tools and their perceptible attributes. The primary aim of this chapter is to evaluate the performance of software clone detection tools from two perspectives. Foremost, clone detection tool assessed by software metrics and subsequent they will be assessed from generated test cases.

The remaining of the chapter is structured as mentioned in the following section . Section 6.1 described code clone notions. In section 6.2 taxonomy of software, clone is discussed. Section 6.3 related to the classification of software clone detection techniques. In section 6.4 evaluation metrics of detection techniques are presented. Section 6.5 entails the literature review. In section 6.6 results are discussed. Further, section 6.7 explores the conclusion with future directions.

## 6.1 INTRODUCTION

Usually, in programming, a software developer copied a code segment from one place and pasted them into another section with extensive modification due to the time constraints. The duplicated cipher is called code clone and the procedure is manifest as code cloning. However, the cut-copy paste activity considers a severe alimentation hitch in source code due to its adverse impingement on accuracy sustainability, and the transformation of application system. The problem with such replicated text is that if a flaw identified in the one segment of code, then it needs corrections in the entire copied segments. Therefore, its detection and analysis is a promising research area due to high maintenance cost [132], and improving the design, quality as well as structure of software system. Several researchers reported that 66% source code is cloned due to replication [58, 133, 146-149] hence, its detection auspicious to locate duplicated code to augment the quality of software systems. The classification of software clone is useful

for enhancing the detection and re-engineering approaches. Since, the copied code in source text may pioneer additional impediments including copyright infraction, replication extension, etc. Over the last decade, the software clone detection has become an emerging research issue in the analysis of software. The adverse impact of cloned software can be addressed by developing a code clone detection approaches. The performance estimation and assessment of these software clone detection tools is a thorny process due to the assorted features of detection methods and limitations of benchmark analogy procedures. The primary objective of this chapter is to measure the portability, scalability, robustness as well as accuracy of software clone detection tools and techniques. Foremost, this chapter begins with the basic introduction of software clone, their types, and detection techniques/tools and subsequently evaluates the performance detection tools and techniques. The potential contribution of this chapter is illustrated below:

- Performance assessment of detection tools and methods in perspective of precision and recall.
- Compare software clone detection tools regarding of portability, scalability, and robustnesss.
- Compare copied code detection techniques in perspective of accuracy named as "precision, recall, portability, scalability, and robustness".
- Compare clone detection techniques on the basis of clone properties.
- Proposed mutation-operators are used for generating variant test cases/ software clones.
- Evaluate clone detection tools by using generated test cases with the existing evaluation methodology.

## 6.2 PERFORMANCE VALUATION METRICS FOR SOFTWARE CLONE DETECTION APPROACHES

Figure 6.1 illustrates the performance evaluation metrics of detection approaches and tools [21, 22, 150]. Figure 6.1 exemplify the performance assessment parameters of code clone detection techniques. Usually, code clone detection tools are evaluated by some performance parameters which are described below.

- **Accuracy:** it indicates how to evaluate the eminence of a copied code. Further, precision and recall are used to compute the accuracy of any detection tools and approaches.
  - **Recall:** it computes the fraction of total number of clones returned by a tool or detection algorithm. A tool which is able to identify all the clone from the source code then that tool should be consider good in recall. However, to calculate the recall of any tool used

the following equation [22, 150, 151].

$Recall = (AC\ X\ 100)/TC$, here, AC means the actual detected clones, and TC specify the total number of copied code occurred in the code. High recall means the maximum number of clones detected by a tool in source text.

- Precision: Precision is relevant to the portion of significant candidate copied code which arises during the detection by detection algorithm.

Precision can be measured using the following mentioned equation [22, 150, 151].

$Preciosion = (AC\ X\ 100)/RC$ ,where RC indicates the absolute clones recoiled by a detection tool.

"A good tool should be able to identify many false positives effectively with higher precision. Higher precision indicates that a tool should be able to find the maximum number of copied codes with higher accuracy (precision)".

- Portability: It indicates the total number of languages supported by a tool. Further, a detection tool should be portable on multiple languages.
- Scalability: It specifies that a tool should be competent to detect copied code from the huge codebase system in a rational instance with the adequate usage of memory. Further, cloning of a code is the most common hitch in the huge system.
- Robustness: It indicates the several expurgation exertions employed on the duplicated code segments in such a manner that a tool can detect different forms of copied code with tremendous accuracy [22].

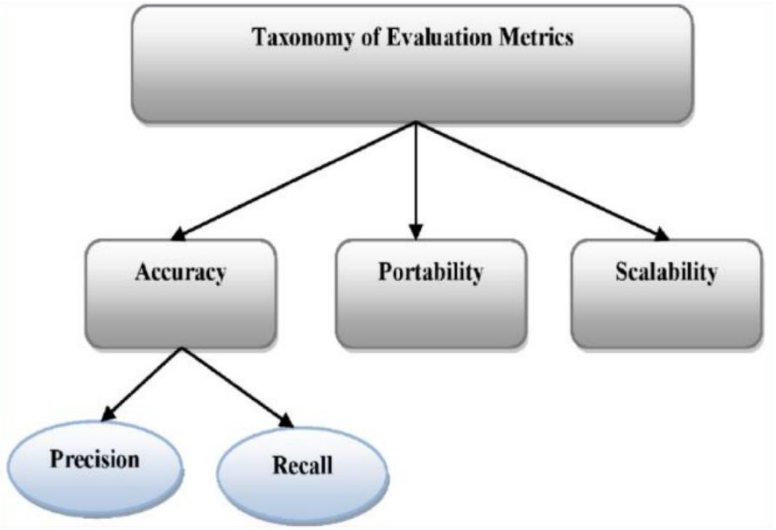



**Figure 6.1:** Taxonomy of evaluation metrics

## 6. 3 RELATED WORK

Since reusing code fragments are generally identical to the original ones, exposing of duplicated source code implicates identified source fragments which are identical. We carried through a significant review of relevant publications to comprehend the expansion, and inclinations in different aspect of cloning research. Since past decades, several detection approaches have been anticipated for duplicated code apprehension. However, earlier techniques and tools are compared by precision and recall. Bellon et al., [152] evaluated six clone detectors with regards to precision and recall. Kaur et al., [150] provided an experimental setup and compared three clone detection tools by precision and recall. Arcelli Fontana et al., [153] provided a comparison of three different software detection tools in perspective of precision and recall as well as analyzing five versions of two open-source systems. Dang et al., [154] presented a survey on the language, techniques, and applications of clone detection tools as well as they also measured performance of clone detection tools regarding precision and recall. Wani et al., [155] explored four software clone detection tools and compared the accuracy of the existing tools. Software clone detection techniques have been compared with respect to different clones properties by Kaur et al., [156]. Recently, Solanki et al., [157] provided a comprehensive review of software clone detection techniques. They also compared clone detection techniques regarding portability, scalability, robustness, and accuracy. Kaur et al., [158] evaluated narration on software clones detection as well as they also thrash out the review on three detection methods namely as "string-based, lexical-based and tree-based" approaches in perspectives of precision and recall. Although, Roy et al., [159] presented benchmark. Though, the aforementioned related work shows that some of the authors compared clone detection techniques concerning portability, robustness and some of them compared clone detection tools by precision and recall. We have found some limitations in existing work:

- A need to detect software clones with high precision and recall.
- A tool is required which can detect syntactic as well as semantic clones.
- A generic automatic and light-weighted clone detection process is required to minimize the computational resources because of existing detection method has various stages.
- A tool should be able to detect many false positive clones so that precision value could be high.
- A tool is required which is better in respect of portability, scalability, and robustness.

Thus, in this chapter, we have evaluated nine code clone detection tools about precision, recall,

portability, scalability, and robustness so that a user can single out according to their requirements. The clone detection techniques have been also compared regarding software metrics as recall, precision, scalability, portability, robustness as well as concerning clone properties of clone detection techniques.

## 6.4 RESULT AND DISCUSSIONS

This section has been divided into four parts. Section A begins with the experimental setup details. Section B focus on software metrics that are used for evaluating clone detection tools and techniques. Section C presents the performance valuation of detection tools and techniques which have been assessed by precision, and recall as well as detection techniques those are measured on behalf of clone properties. Section D proposed literals/mutation-operators that are used to generate test cases or software clone with the help of proposed literals. Further, software clone detection tools and techniques evaluated by using generated variants of test cases or software clones.

### A. Experimental set up

The proposed work used an open source Gantt Project and window 7, 4 GB RAM, 1.90 GHz processor for valuation of code clone detection tools and techniques. Further, nine software clone detection tools have been evaluated from three parameters which are depicted in Figure 6.1. Table 6. 1 shows the evaluation outcomes of clone detection tools which are assessed by accuracy as “precision; recall”, and various forms of clones [120].

### B. Performance metrics

Performance of code clone detection tools is calculated from three metrics which are illustrated in Figure 6.1.

- Accuracy: This metrics entails the quality of code clone detailed by a tool. The accuracy can be assorted into two types which are described below.
- Precision: It means the total number of corrected identified clones from the candidate’s clones which are returned by a tool.

PMD has the highest recall while Bauhaus is good in precision value. The maximum number of corrected clones returned by PMD from the candidate clones. CloneDigger returns a huge amount of “false positives” due to least precision value. “SDD, NICAD” has similar recall

percentage while iClone has the minimum recall value. Farther, SDD, NICAD, CPD and iClone tools have no differences in their recall value. The high recall means a tool should be able to perceive maximum number of clones with less number of false positives. Although, few of the tools named as SDD, CCFinderX, and CPMiner tools have slight distinction in their accuracy rate. However, the copied code can be detected using many detection tools and techniques. Hence, a comparison of these code clone detection tools is necessary by their performance so that a user can choose the right tool as per their requirements. Table 6.2 illustrates scalability, portability, and robustness of code clone detection tools.

**Table 6.1:** Comparisons of software clone detection tools

| Approach | Tools | Clone types | Precision | Recall |
|---|---|---|---|---|
| **String-based** | SDD | Type 1,2,3 | 0.24 | 0.22 |
| **String-based** | NICAD | Type 1,2,3 | 0.17 | 0.22 |
| **Lexical-based** | CPD | Type 1,2 | 0.11 | 0.24 |
| **Lexical-based** | iClone | Type 1,2,3 | 0.26 | 0.2 |
| **Tree-based** | CloneDigger | Type,1,2 | 0.03 | 0.56 |
| **String matching** | PMD | Type, 1,2,3 | 0.46 | 0.59 |
| **Lexical-based** | CCFinderX | Type,1,2 | 0.56 | 0.51 |
| **Lexical-based** | CPMiner | Type,1,2 | 0.41 | 0.48 |
| **AST-based** | Bauhaus | Type,1,2 | 0.81 | 0.49 |

As Table 6.2, explores several tools which are based on distinct methods or techniques for code clone detection, though they all have similar functionality of code clone detection. In this performance evaluation, seven tools are calculated from portability, scalability, and robustness. Each of them is based on distant methodology of code clone detection as CCFinderX used token-based approach while NICAD is based on text-based approach.

**Table 6.2:** Comparisons of portability and scalability of clone detection tools

| Approach | Tools | Portability | Scalability | Robustness |
|---|---|---|---|---|
| **String-based** | NICAD [160] | Y(highly) | Y(highly) | Y(highly) |
| **String matching** | PMD [124] | Y(highly) | Y(highly) | X(limited) |
| **Lexical- based** | CCFinderX [125] | Y(highly) | X(limited) | X(limited) |
| **Lexical- based** | CPMiner [60] | X(limited) | Y(highly) | X(limited) |
| **AST-based** | Bauhaus [65,127] | Y(highly) | X(limited | Y(highly) |
| **PDG- based** | GPLG [74] | X(limited | Y(highly) | Y(highly) |
| **Hybrid** | ConQAT[161] | Y(highly) | Y(highly) | X(limited |

### C. Performance evaluation of code clone detection tools and techniques

SDD: Similar data detection (SDD) is a string-based method which is proposed by Seunghak et al., [161]. It is used as a plug-in in Eclipse. This tool is used to identify “type-1 to type-3 code clones” from a large scale system with high performance.

- **NICAD:** Roy et al., [159] proposed a string-based tool namely as Accurate Detection of Near-Miss Intentional Clones (NICAD). Though, NICAD is a hybrid approach –based (text-based and abstract syntax tree ) tool Further, NICAD is able to identify type-1 to type-3 clones from a large scale system.
- **CPD (Copy Paste Detector):** Sourceforge community proposed a tool namely as CPD. This tool is an add-on of PMD.
- **CloneDigger:** This tool is developed by Peter et al., [162].

(Programming Mistake Detector): It is designed by Sourceforge community in 2000. PMD is based on the Rabin-Karp string search algorithm for detecting code clone.

- **CCFinderX:** Kamiya [125] explored a token-based tool named as CCFinderX. Further, it is divergent of CCFinder [59]. Although, it can detect "type-1 to type-2 clones" inadequate to identify copied code clones from a generous software.
- **CPMiner:** Li et al., [60, 61] developed a token-based tool for the detection of 1-2 software clones from a huge application system.
- **Bauhaus:** It is a deviation of "CloneDR" with few variations. "Bauhaus" [65, 127] developed a tool namely as "ccdiml". The proposed tool used the "abstract syntax tree"(AST) for the exposing of "type-1 to type-2 code clone.

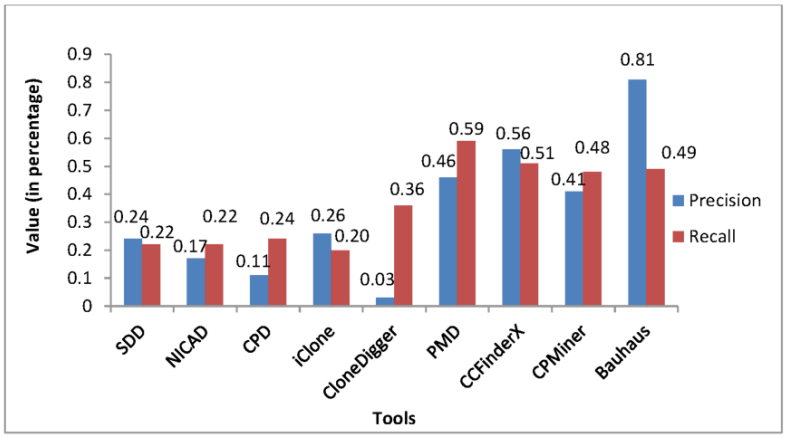



**Figure 6.2:** Comparisons of clone detection tools w.r.t precision and recall values

**D. Generation of test- cases using mutation operators**

Figure 6.3 shows the mutation operators for test-cases or software clones. There are four types of code clone. Type-1 clones can be generated by using several mutation operators as mCW remove white space, mCW- changes in blank spaces, mCC- changes in comments, mCF- changes in formatting and mCNWs- changes in new line spaces. Using mutation operator's type-2 clones can be created as mARV- arbitrary renaming variables, mRPE- replacement of parameters with expressions, mARDT- renaming data types and mARL- arbitrary renaming of

literals. Type-3 clones generated by mSDL- small expunging within a line, mSIL- small insertions within a line, mMLs- modification in the whole line, mAOR-changes in arithmetic operators and mDSV- variation data statements. Type-4 clones created by using mROS-reorder the statements, mCR- replace one type of control statement with another type of control statement, mCSS-constant substitution.

Figure 6.4 depicts the layout which is based on mutation operators for generating test-cases/software clones. The proposed layout is classified into four types as S1, S2, S3, and S4 and by using mutation. Layout S1 also has four types as S1 (a) mCW remove white space, S1 (b) mCC- changes in comments, S1 (c) mCW- changes in blank spaces, S1 (d) mCNWs-changes in new line spaces and S1 (d) mCF-changes in formatting. The second layout is S2 which has been classified into four types. S2 (a) mARV- arbitrary renaming of variables systematically, S2 (b) mARV- arbitrary of renaming variables non-systematically. S2 (c) mARDT- renaming of data types S2 (d) mRPE- replacement of parameters with expressions.

**Table 6.3:** Comparisons of clone detection approaches w.r.t properties and metrics

| Properties | Text-based | Token-based | Tree-based | PDG-based | Metrics-based |
|---|---|---|---|---|---|
| Transformation | Removes whitespace and comments | The source code is divided into tokens | AST is generated from the source code | PDG is generated from the source code | To find metrics values |
| Representation | normalized source code | In the form of tokens | Represent in the way of an abstract syntax tree | Set of program dependency graph | Set of metrics values |
| Comparison based | tokens of line | Token | Node of tree | The Node of program dependency graph | Metrics value |
| Computational complexity | Depends on algorithm | Linear | Quadratic | Quadratic | Linear |
| Refactoring opportunities | Good for exact matches | Some Post processing needed | It is good for refactoring because of find syntactic clones | Good for refactoring | Manual inspection is required |
| Language independency | Easily adaptable | It needs a lexer, but there is no syntactic knowledge required | Parser is required | syntactic knowledge of edge and PDG is required | Parser is required |
| Portability | High | Medium | Low | Low | Relative to defined metric |
| Accuracy | High | Low | High | High | High |
| Robustness | Low | Limited | High | Medium | Medium |
| Scalability | Relative to the comparison algorithm | High | Relative to the comparison algorithm | Low | High |
| Clone types | Type-1 to type-2 | Type-1 to type-2 | Type-1 to type-3 | Type-1 to type-4 | Type-1 to type-3 |

**Table 6.4:** Summary of evaluation metrics

| Approach | Precision | Recall | Scalability | Portability | Robustness |
|---|---|---|---|---|---|
| **Text/String-based** | H(High) | L(Low) | Rely upon comparison methods) | H(High) | L(Low) |
| **Token-based** | L(Low) | H(High) | H(High) | M(Medium) | M(Medium) |
| **Syntactic/Tree-based** | H(High) | L(Low) | Rely upon comparison methods | L(Low) | M(Medium) |
| **Syntactic/Metric-based** | M(Mediu m) | M(Medium) | H(High) | L(Low) | M(Medium) |
| **Semantic/ PDG-based** | H(High) | M(Medium) | L(Low) | L(Low) | H(High) |
| **Semantic/Hybrid-based** | M(Mediu m) | H(High) | M(Medium) | M(Medium) | H(High) |

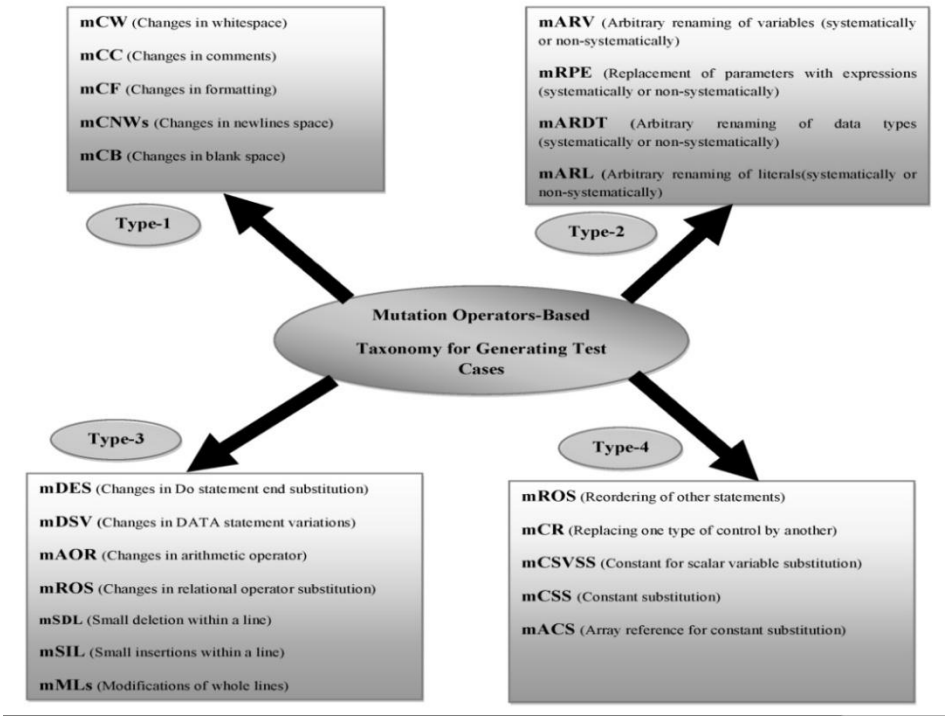



**Figure 6.3:** Taxonomy of mutation-operators for generating variants types of test-cases (software clones)

The S3 layout has been assorted into five types as S3 (a) "mSIL- small insertions within a line, S3 (b) mSDL- small deletions within a line", S3 (c) mMLs- modification across the entire line as insertion of statement, S3 (d) mMLs- modification in the whole line as deletion of statement, S3 (e) mMLs- modification in the whole line. S4 layout can be segregated into four types as S4 (a) mROS-reorder the declaration statements, S4 (b) mROS-reorder data-independent statements, S4 (c) mROS-reorder the data-dependent statements S4 (d) mCR- replace one type of control statement to another kind of control statement. The valuation of detection approaches,

and tools in perspective of "precision and recall" on the basis of scenario or layout which is shown in Table 6.5. We developed a scenario for evaluation by using mutation operator-based editing taxonomy. At first, the editing scenario generate various type of software clones using mutation operator-based editing taxonomy , secondly created variants of software clones used for evaluation of detection tools.

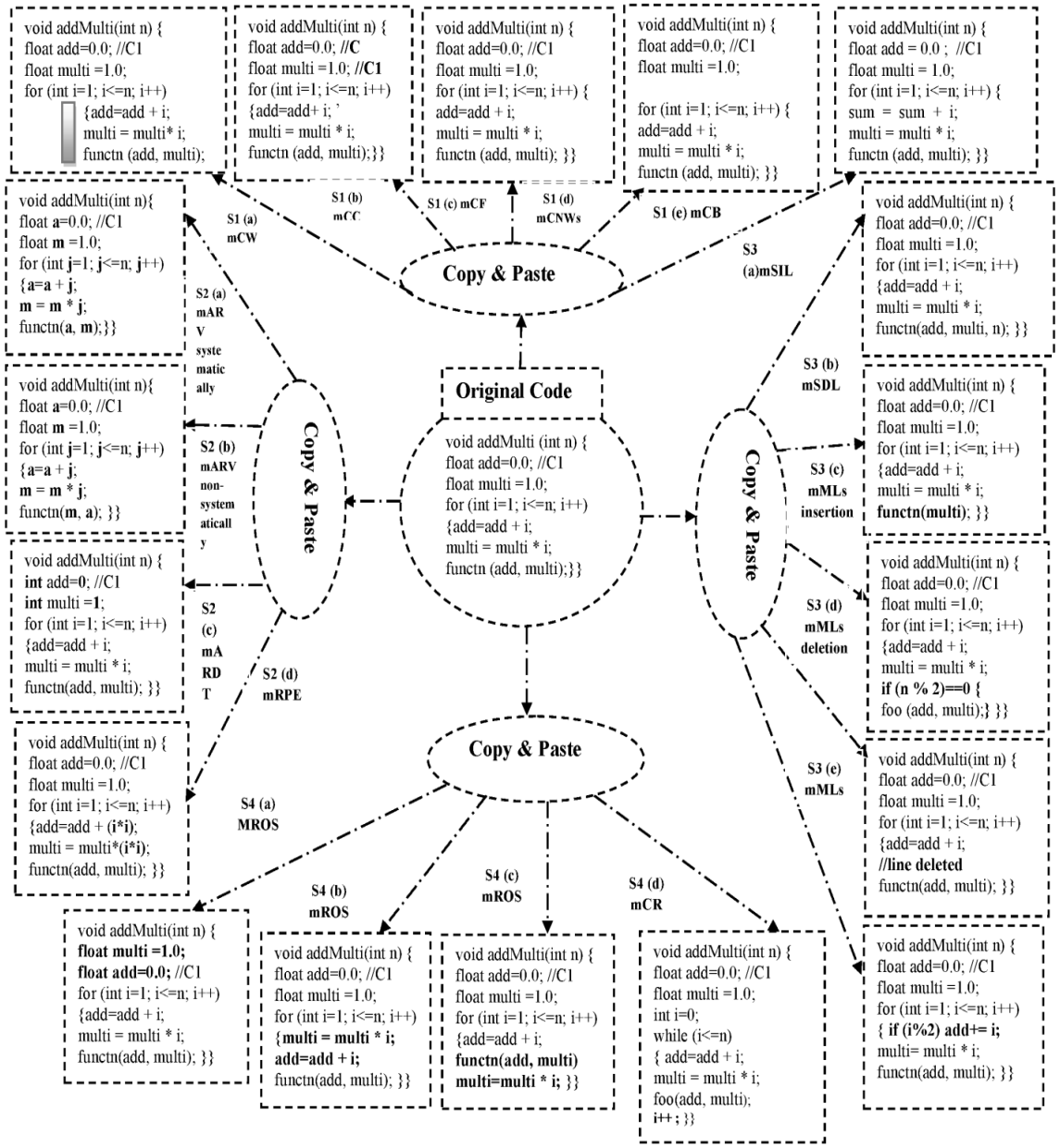



**Figure 6.4:** An example of test-cases (software clones) generated using mutation-operators

Overall performance of detection tools in perspective of precision is shown in Figure 6.5. The clone detection tools have been compared by using the generated test-cases or software clone with the existing methodology which uses only two or three variants of software clones while we have evaluated detection tools by generating various clones or test-cases which are created with the help of mutation-operators.

**Table 6.5:** Comparisons of clone detection tools w. r .t test cases generated using mutation-operator

| Approach | Tools | Expected precision w.r.t scenario | Expected recall w.r.t scenario | Clone types |
|---|---|---|---|---|
| **Text-based** | NICAD[(Roy and Cordy, 2008) | *(Good) | #Excellent) | (*) In detection of type-1 to type-3 |
| **String matching** | PMD(Sourcefourge community,2007) | *(Good) | *(Good) | (*) In detection of type-1 to type-3 |
| **Token-based** | CCFinderX (Kamiya. T , 2008) | @(Medium) | @(Medium) | (@) In detection of type-1 to type-2 |
| **Token-based** | CPMiner(Li et al., 2004) | @(medium) | *(Good) | (@) In detection of type-1 to type-2 |
| **AST-based** | Bauhaus (Bellon. S ,2002; Raza et al., 2006) | @(Medium) | *(Good) | (@) In detection of type-1 to type-2 |
| **PDG- based** | GPLG(Liu et al., 2006) | @(Medium) | @(Medium) | (@) In detection of type-1 to type-2 |
| **Hybrid** | ConQAT(Hummel et al., 2010) | $(Low) | @(Medium) | (@) In detection of type-1 to type-2 |

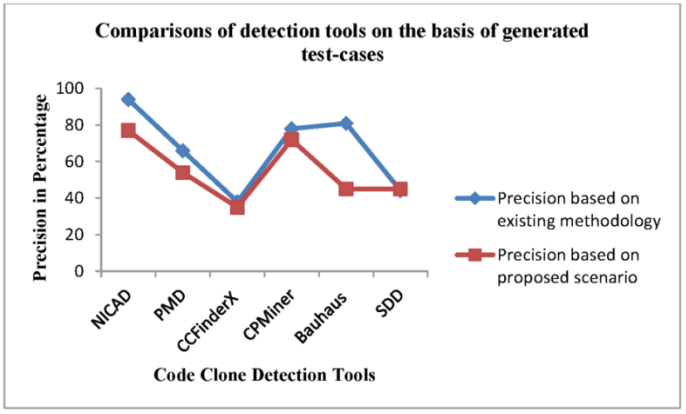



**Figure 6.5:** Performance evaluation of clone detection tools on the basis of generated test cases with the existing methodology

## 6.5 CONCLUSION AND FUTURE DIRECTIONS

In this chapter, code clone detection tools evaluated into two different dimensions. Foremost, to analyze the precision, recall, scalability, portability, the robustness of detection approaches and tools, and subsequently to assess the performance of "code clone detection tools" in respective

of generated test cases. The proposed work compared token, string, tree, PDG as well as the hybrid approach based code clone detection tools. Although, as per the authors best knowledge, there has been no performance evaluation has been done which can compared code clone detection tools and techniques regarding accuracy, portability and scalability, and robustness. As per the study reports which shows that Bauhaus has the maximum precision percentage, but it is not competent on huge codebase. PMD unable to detect type 3 clones but it has good recall value in comparisons of other tools. NICAD is highly portable, scalable, and robust with low accuracy while Bauhaus and PMD are better tools regarding accuracy. It is thorny to conclude a luminous "victor" of this emulation because all tools have their vigor and impuissance and hence, are applicable for distinct task and contexts. Nonetheless, the assessment illuminates on some particulars that were not known earlier. Though, there are several considerable notches to note when viewing at the outcomes of the evaluations:

- The lexical and the string-based techniques are astonishingly similar such as in Table 6.1 two "text-based techniques" have the similar accuracy ratio, and two lexical-based approaches have approximately the equivalent recall rate.
- The tree and text-based tools have higher recall.
- "Bauhaus and CCFinder" have good or higher precision percentage.
- Text-based tools have higher portability, scalability, and robustness.
- The CCFinderX is portable on multiple languages but unable to detect type-3 clones on the large system.
- The "CPMiner" can exert on the generous code but inadequate to identify "type-3 clones" which are developed in multiple languages.
- Bauhaus has high portability and robustness but unable to identify clones in huge software system with adequate memory as well as time.
- The GPLAG is not portable.
- ConQAT tool is based on a hybrid approach which can identify only "type-1 and type-3 clones".

The overall comparison of copied code detection techniques concerning evaluation metrics is shown in Table 6.3 and Table 6.4 shows the comparisons of clone detection techniques in respect to metrics and properties. The high precision means a tool should be able to detect fewer false positives. String- based techniques has low robustness and low recall because it identifies only type-1 clones with high precision. Scalability of line-based and tree-based methods depends on comparisons algorithms. Token-based techniques have low precision and

medium portability due to transformations/ normalizations rules, but it has the high recall because it detects most of the clones. Token-based methods have high scalability when they use suffix-tree algorithms. The syntactic approach can be assorted as Tree-based and metric-based. Tree-based methods have high precision because of structural information, but it has the low recall and medium robustness due to which it cannot detect clone. Metric-based techniques have medium precision, robustness due to two code fragments having identical metric values. The portability of metric-based methods is low because it needs a parser to generate metric values, but it has high scalability due to begin end blocks of metric values are compared. The semantic-based approach can be categorized as graph-based and hybrid. PDG (program dependency graph) is an example of the graph-based approach. PDG has high precision and robustness due to both structural as well as semantic information. The recall of PDG is low because it cannot detect all clones. PDG-based techniques have low portability due to PDG generator and low scalability because sub-graph matching is expensive. The hybrid-based methods have medium precision and high recall, robustness because of it can detect type-1 to 3 clones. It has medium portability due to which kind of hybrid techniques are used. Thus, the objective of this chapter to evaluate the performance of code clone detection tools using distinct parameters. The current aim of this evaluation to compare code clone detection tools/techniques are to reveal the usefulness and effectiveness of clone detection tools. On the basis of their perceptible features, user can easily know which tool is sophisticated for their requirements. As per the author's best knowledge there is standard benchmark available in the literature for comparative analysis. Thus, we proposed a common layout for assessing clone detection tools in future. By using proposed layout evaluation and comparisons will become more competent and protected.

# CHAPTER 7
# USES OF MUTATION OPERATORS IN CODE CLONES

The objective of this chapter is to uses mutation operators for code cloning. Although, this chapter is divided into three major sections. Section 7.1 proposed an editing taxonomy for creating distinct types of code clones. Section 7.2 presents a hypothetical scenario for evaluation code clone detection tools. Moreover, the hypothetical scenario generated using mutation operator-based editing taxonomy. Section 7.3 related to mutation operator-based an automatic structure for injecting and detecting code clones. The rest of the sections veil the elementary terms and background context for testing and analysis of mutation, the concerned work over the mutation testing, come up with redact's nomenclature of mutation operators for code cloning research and at last, conclude forthcoming work.

## 7.1 AN EDITING TAXONOMY OF MUTATION OPERATORS FOR CLONE GENERATION

Over the past decade, many code clone detection methods and tools are introduced by numerous scientists. Though a large number of tools are concerned in an exceedingly perceptible assessment, and a lot of efforts are made to assessed and analyze numerous contemporary tools systematically. Furthermore, little experiential analysis has been accomplished by studying the performances of those tools. The present study signifies that several facets might influence the authenticity of the results of those evaluations. To abolish the performance of this facet; an outline of the mutation-based nomenclature for various types of code clones is presented in this section. The proposed classification would facilitate to valuate clone detection tools and methods through factual study.

### 7.1.1 Introduction

Customization of the program text is accomplished by copying-pasting from one place to another place and frequently it implicates code replication. The programs fragments which are similar are known as code clone meanwhile the development are called code cloning. The preceding study exemplifies that a considerable part of software systems have replicated a significant extent (20-59) in their code [15, 17, 18, 19, 56, 111]. Code cloning discourage the refactoring due to bug propagation (if an error unearth in one segment of code, then it corrected

for the same bug into the entire reproduce fragments) [61]. Moreover, the endeavors are further augmented due to the duplicated code when we enhance a software system [53]. Although, various code clone detection approaches have been detailed earlier, a recent survey of those methods has been discussed by the authors [111]. Several relative evaluation and computations of these approaches have been performed by many scientists to study their efficiency from distant aspects [52, 151, 163, 164, 165]. Nonetheless, since software clone detection approaches has different attributes, it's relatively challenging to evaluate peculiar tools used in clone detection and thus these studies include comprehensive amplification to the software for analysis of clone detection [68]. In recent days, Rattan et al. [20] has given a reasonably review on software of clone detection, while Roy et al [12], present a qualitative valuation for tools and clone detection methods. Belton et al., [152], exemplified a inclusive assessment open source software for six different clone detectors in single out software clones in C and JAVA. In spite of thorough concept, only a partial fragment of clones were condensed, and As a result many other aspects were identified. [68]. particularly, faulty assessment is annoyed by the fact that there is no general assessment standard. It is intricate process to get a general standard as every perspective has its own specification, drafted for distinctive intents.

In current section, we have depicted some of the hypothetical altering scenarios, which are replica of generic changes to the copied and pasted code. This research is an endeavor to use a code clone taxonomy based on a mutation that is wide employed by the testing community. Meanwhile, its use as a stage which exhibited a realistic evaluation entity for quantifying and expanding several test suites [166]. The testing based on mutation methodologies has an impressive role in assessing detection tools for code clone, if mutation operators set has been summarized that imitable cloning study for a nominal language.

### 7.1.2 Overview for Testing or Analysis of Mutation

Software testing has many important fractions such as empirical analysis of test approaches. DeMillo et al. [167] and Hamlet [168], proposed most competent and potent methods for software testing that is mutation testing. Mutation testing instigation uses to construct the test data. Though, mutation testing faults create several variants for program and still forefront in source code [169]. Apart from this, mutation possible advantage as the mutations operators can be exemplified precisely and thus a specific, faulty seeding process can be illustrated. [166]. For initiating the alternative of program text, these faults can be included repeatedly or manually by engineers. Typically, we study an self-drive developed adaptation which is formed by

implementing the operator in the program text. these types of employed operators are recognized as mutation operators. The generic process is termed as the mutation and the emerging flawed renditions are called mutants.

### 7.1.3 Related Work

In 1971 Lipton [13] proposed the history of mutation testing using a student paper. Although, first time DeMillo et al. [167] and Hamlet [168] introduces the mutation testing field with his papers in the late 1970's. They provided first source code mutation-based study. Even they are the main persons, who majorly use mutants for calculate test-case amplitude while Offutt and others [169] only analyze comprehensively mutation testing, the same author [170] later proposed first assessment exertion, which was in his early phase of expansion of mutation testing in 1989 and same work defined the circumstances and research accomplishments in the fields of mutation testing.

Some important mandatory proposals for mutation testing was persistent by Offutt et al. [171], that's a test dataset which analyzes elemental faults commenced by mutation will identify synthesized faults, namely, the blending of numerous elementary faults. Frankl et al. [172], Thevenod-Fosse et al. [173], and Hutchins et al. [174] performed experimentation using flawed variations of programs, and since then a lot of scientists have reviewed this system. Frankl et al. [172] utilized Nine-Pascal programs while each application having one current defect. Hutchins et al. [174] uses those programs, which included 7 programs with 130 hand-seeded drawbacks. Four small C-programs were used using mutation operators, which were intended for repeated errors in the program by Thevenod-Fosse et al [173]. Generally, The texture construction of a large "test pool" of test experiments is made after these experiments. The same author also employed mutant generation in their research to create a research adaptation, although this was initially possible as part of the test approach. Kim et al.. [175], Memon et al.. [176], Andrews et al., [177], and Briand et al., [178] were used the same method. Although, Chen et al. [179] were used hand-seeded faults and generated mutants meanwhile he illustrated that the exaggerated defects by hand are fragments of potential flaws, but they have not considered it in their research. Muthra [180] proposed their software testing projects using mutation analysis. Analysis of mutation, to evaluate the properties of the data used to review the program code, has several light comparative deviations of the related program text to evaluate. Some specific constraints of a language system have been illustrated by King and Offutt [38], and they also explicates how it alters and restraints a mutation based background where testing exists.

Andrews et al. [177] appointed a similar type of program for the mutation production program to create mutants for the code written in C programming language. The mutation production program of Andrews and Zhang [34] was used by Andrews et al[166] for the creating mutant of a subject program and compares the flaws by explaining the power of test suits on hand-seeded, real-world errors and automatically generated errors. Agrawal et al. [46] proposed and discussed the literature in detail with indexed seventy-six operators of mutation. Agrawal et al., [181] briefed the proteum mutant generation system, which is an expanded system, executing 108 operators including almost each operator.

The authors [182] have briefed a review with reference to the expansion of mutation testing. Similarly, in the detection process of code clone, programmers customize segments of program with copy / pasting, potentially or with slight modifications. These slight modifications in code is called code clone, and the process of replication code is called code cloning. However, Roy and Cordy [183] first applied mutation perception in cloning detection research for the first time, in a brief manner. They expected a theoretical editing scenario for different types of clone and based on that scenario, also introduced a control structure based on for the evaluation and contrast of identification methods [184]. According to the author's knowledge, no work has been done in the clone detection field using mutation tests. In addition, no experimental study has unmodified the operation of mutants, as far as our facts are frightened. The author's contribution is as follows.

- Used of mutation operators in Clone Detection Research era.
- An editing nomenclature is expected which is based on mutation operators.
- To make several types of code classification, using this classification.

### 7.1.4 Proposed Editing Classification for Cloning

The revelation of the clone is essentially vague in literature. Code clone is specified by Baxter et al., [64] as a fragment of code which is distinguished by some characterization of similarity. Kamiya et al., [42] further described code clones as piece of a code which is “identical” or “similar” to each other, where “similar” was not detailed by them but by “identical” they imply exact facsimile. According definition given by Burd et al.,[164], a fraction of the code is phrased in the form of a clone, if two or more expressions in that code section are present with or without "slight" modification.. In most cases, we are studying for "clones" which are created as a result of optimization/facsimile/paste processor through programmers. It starts with a hypothesis and incorporates it as the foundation mechanism for the top-down claim of the code

clone, which has been defined as classification in a process type, which is attempted in a predefined concept of clone given by developer [183]. It is not just a complete speculation, as if considered suspiciously, we would stumble on that it is consequent from the clone analogue [53, 168, 174, 179, 184] which are extorted and copied from the immense noted works, along with the clone types [163], clone classifications [86,17], thorough review of programmer copy/paste behavior [40].

The availability of a set of fake operators is defined as the main apprehension for a mutation-based trial. As far as the mutation operator has not been used for cloning the code, while many mutation generators are available in many languages to create clear flaws [166]. To maintain mutation analysis in this section, an edit-out classification of different types of code clones has been presented. Apart from this, a set of mutation operators can also be taken for code cloning, with the aid of proposed editing-outside classification. Mutation operators are consulted in the mutation test study to modify the source code to present possible errors. In the Code Clone Research era, duplicate operators are those removal behaviors that make new fessimid codes by replication/pasting. This section brings about thirty seven different (sub) category of mutation operator in Table 7.1 by elevating existing mutation operator from the literature [169, 180]. For cloning, thirty-seven types of mutation operators have been revealed by each of the sixteen clone types introduced by Roy and Cordy [183].

In order to generate potential errors in an original code, mutation methods can be sorted into data, design, layout and control mutation expiring activities, whereas the estimated classification for the mutation operator is a widespread. We can ensure that, through the estimated classification of mutation operators, different code clone aspects can be originate

Mutation operators have been employed to produce clear defects in many languages and software testing while we face limitation in mutation operators for code cloning. Table 7.1 given a wide classification of mutation operators based on the editing classification shown in Figure 7.1.

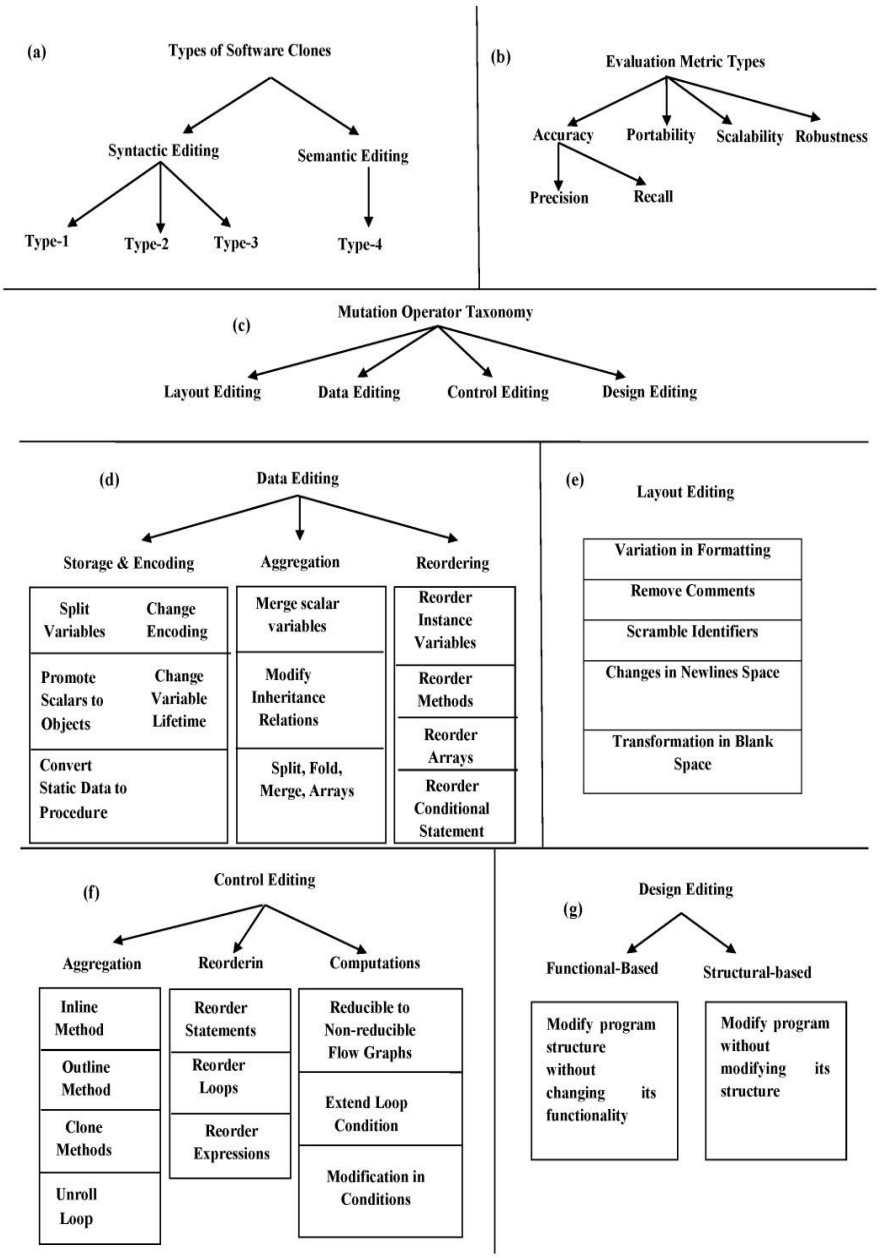



**Figure 7.1:** (a) Show category of code clones, (b) Illustrates the evaluation metrics, (c) demonstrates distinct operator for mutation. (d) Data-Based Editing Mutation Degrees operator (e) "lexical-related expurgation activity mutation degrees operator (f) control-based expurgation mutation degrees operator and (g) design-based editing activities mutation degrees operator.

**Table 7.1:** A mutation operator-based generic taxonomy for software cloning

| Name | Arbitrary Editing Activities | Types of Clone |
|---|---|---|
| **mRW** | Removes whitespace | **Type-1** |
| **mCNWs** | Changes in newlines space | |
| **mMCs** | Modifies comments | |
| **mVF** | Variations in formatting | |
| **mTBs** | Transformation in blank space | |
| **mARV** | Arbitrary renaming of variables (systematically or not systematically) | **Type-2** |
| **mARDT** | Arbitrary renaming of data types (systematically or not systematically) | |
| **mARL** | Arbitrary renaming of literals(systematically or not systematically) | |
| **mDEs** | Changes in the Do statement end substitution | **Type-3** |
| **mDSv** | Changes in DATA statement variations | |
| **mGLs** | Changes in GOTO label substitution | |
| **mRSs** | Changes in RETURN statement substitution | |
| **mSAn** | Statement analysis | |
| **mABs** | Changes in absolute value insertion | |
| **mAOr** | Changes in arithmetic operator | |
| **mLCS** | Changes in logical connector substitution | |
| **mROs** | Changes in relational operator substitution | |
| **mUOI** | Changes in unary operator insertion | |
| **mLIi** | Little inclusions within a line | |
| **mLDl** | Little deletion within a line | |
| **mILs** | The inclusion of one or more lines | |
| **mDLs** | Deletion of one or more lines | |
| **mALs** | Adaptation of all lines | |
| **mROS** | Reordering of other statements | **Type-4** |
| **mSCRs** | Substitute one type of control by another | |
| **mAAs** | Array reference for array reference substitution | |
| **mACs** | Array allusion for constant substitution | |
| **mASs** | Array allusion for scalar variable substitution | |
| **mCAs** | Changes in constant for array reference substitution | |
| **mCNs** | Changes incomparable array name substitution | |
| **mCSs** | Constant substitution | |
| **mCSVSs** | Constant for scalar variable substitution | |
| **mSAs** | Changes in a scalar variable for array reference substitution | |
| **mSCs** | Changes in scalar for constant substitution | |
| **mSRCs** | Changes in source constant substitution | |
| **mSVs** | **Changes in scalar variable substitution** | |

### 7.1.5 Conclusions and Future Work

This section aims to present a comprehensive study on mutation testing, analysis and by using this assessment to propose a layout of mutation -based taxonomy for code clones. Further, by using this taxonomy to derive mutation operators for code cloning. The future research hypothesizes, of course, to pioneer a mutation-based framework for completely assessing code clone detection tools and methods. Moreover, the anticipated framework efficiently measures and contrasts the clone detection tools which were based on different detection approaches.

## 7.2 A SCENARIO BASED ON MUTATION OPERATOR FOR EVALUATING SOFTWARE CLONE DETECTION TOOLS AND TECHNIQUES

Over the last decade, many code clone detection techniques have been acquainted by several researchers. Although, every method have their tunable parameters and they depend on various dimension and sub-dimensions which make them thorny to do the relevant study. Thus, it is necessary to propose a hypothetical scenario based on mutation operator to evaluate specific types of identification tools and techniques. This section presented a hypothetical scenario for many assessments using the simulation operator-based editing classification for code clone detection tools which are based on distinct types of detection techniques. In addition to this, the existing evaluation criterion is extended by the hypothetical scenario which is explicitly represented by the analysis of results.

### 7.2.1 Introduction

Probably reusing the code segment with some changes is the continuous activity in the process of software development. As a result, the replication of code we are known as code clone and this practice by code cloning. Code clones pioneer bugs in the software system and augment maintenance exertions. Previous literature [15, 17, 19, 56, 72, 111] states that due to replication, 20-59% code is duplicate code. A copyed segment is shown in a portion of the code, then it should be made valid for the same flaws in the entire software. Propitiously, many tools and technique for code clone have been proposed, and some assessments have been abstraction related to them. Presently, Rattan et al., [20] provided a systematic review on code clones while Roy et al., [147] presented a qualitative contrast and evaluation tools and approaches for code clone detection. Further, authors have also provided theoretical editing scenario-based evaluation techniques for Clone identification [147]. The six clone detector tools evaluation which is based on c and Java presented by Bellon et al., [152]. Additionally, many latent abstractions have been reported that clone detection tools have been evaluated using precision, recall, portability, and computational complexity [22]. This section presents an extended analogy of code clone detection techniques and tools which are accessible presently using aspect. The main intent of this section is to point out the significant potency and constraints of individual tools and methods by their aspects and proposed mutation operator-based editing hypothetical scenarios. The primary objective is to provide an extensive classification of

existing techniques and its potential to analyze "legitimate code clone." In addition to this, to create a mutation operator based composed classification which is used to spawn distant code clone forms. Moreover, these scenarios are usage to evaluate numerous code clone detection methods and tools. By considering the editing scenarios, it is easy to analyze proposed work which as a huge range of distinct features when evaluated with prior reviews. Hence, the prime objective is to presents a successful estimation which is adequate, realistic and potential proof as we want an evaluation in which there is potential for future and is not only an implementation tool for the present

### 7.2.2 Attribute-Based Comparisons of Clone Detection Techniques and Tools

Figure 7.2. The dependency of language on second attribute is a category of program rummage that depends on an exact technology to generate intermediate representation. The third attribute, which shows that only some tools support object-oriented language, is related to language support of the device. The fourth attribute, indicates how the clones return as a clone class, a clone pair, or both, shows the clone relation. The fifth feature is related to the type of technique of the clone. Normally, the software clone is classified into four categories.

Type-1 to Type-3 text is related to the same equality, while type-4 clone has functionality with a different structure, are the functional clone. The sixth attribute is attached to the granularity of the returned clone such as free or fixed. The seventh attribute related to the performance stage for which the tool is viable. For example, many tools that run on Linux or some other windows run. In other words, the tool platform may or may not be dependent or independent. The eight attribute provides information about the comparison of algorithms, which identifies different comparison algorithms, which are used for more detail as described in Figure 7.2, in data mining, finger-printing and hash values of clone detection research. The ninth attribute shows computational complexity that leads to an identification technique to detect clones in large systems. Generally, complexity computation based on comparison of algorithms and types of changes. The tenth attributes is a comparative granularity, in this approach, the clone detection technique is applied at different levels, such as identification technology based on text, token, syntax tree for more detail, see Figure 7.3.

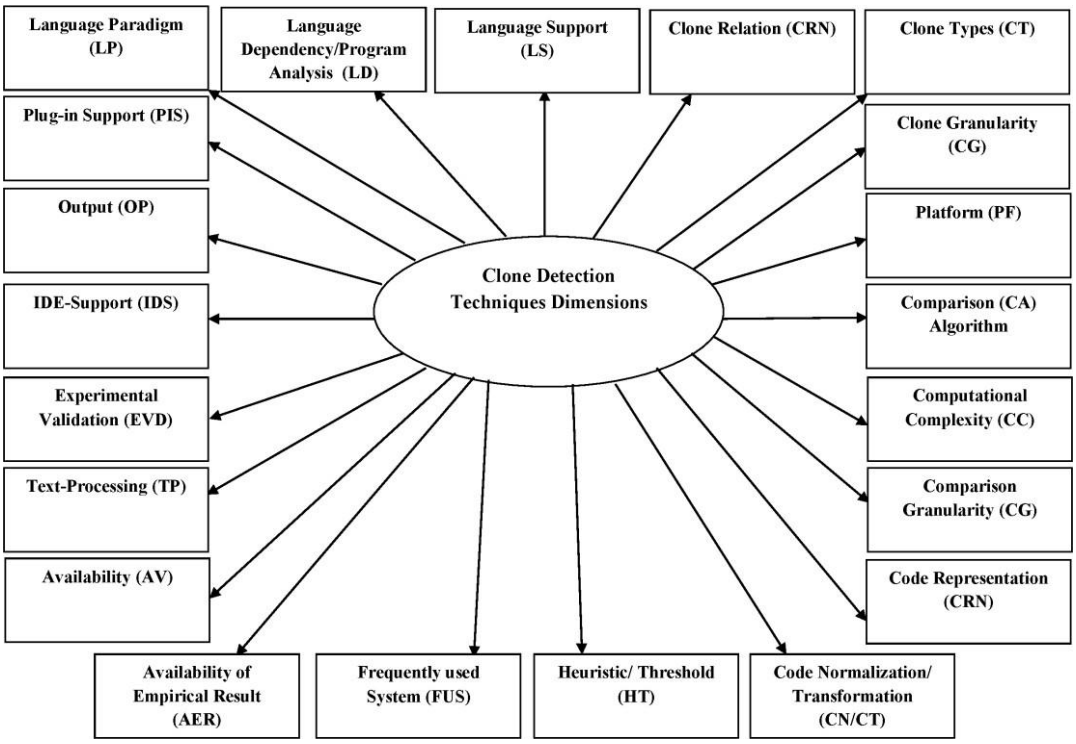



**Figure 7.2:** Clone detection techniques properties

The eleventh attribute represent the code representation aspect, which gives internal code representation after filtering, normalization and transformation. The twelve attribute are about the code generalizations /transformation, which are related to changes in whitespaces, comments; See Figure 7.3 for more information. The thirteenth attribute, which is used by a exact technique, is relevant to the heuristics / threshold and specifies whether there is a exceeded intensities and heuristics like code similarity, difference size (Figure 7.3).The often used system is our 14th attribute, indicating that common methods like JDK and Kernels have been used in verification. The fifteenth attribute shows the availability of empirical results whether or not the validate results are available, if the results are available, then other researchers may be adept to use them. Sixteenth attribute is about, the tools are open source or commercially access. Seventeenth attribute relates to text-processing as pre-post-processing / glamorous printing instead of any other required general filtering. The eighteenth attribute is about experimental verification. This reflects the type of verification recognized by a technique. The nineteenth attribute gives information about support of IDE, that tools are supported by IDE or not. The twentieth attribute is about the output, and it refers to what type of output is textualized or visualized by the tools. Twenty-one attribute refers to plug-in support, this indicates whether the device is part of an IDE or not.

### 7.2.3 Mutation Operators-Based Editing Taxonomy for Software Clones

Mutation testing is a form of testing for software that is used to make variations of flaws without using a different behavior in the source code. These defects can be assimilated by manually or repeatedly. Offutt [14] states that the errors introduced in the source code are based on some well-defined conventions called mutation operators.

Figure 7.4 shows Type-1 and Type-2 code clones that are created using mutation operators.Type-1 can be generated by adding or deleting whitespaces, new-line spaces in source code, adopting comments, changes in formatting, and some amendments in blank spaces. In the meantime, by changing the name of variable, literal names change whether systematically or not and some transforms in parameters, we can generate the Type-2 clones. Figure 7.5 is about code clone of Type-3. Generally, Type-3 code clones can be included by adding / removing new lines, changes in the relation operator, logical / arithmetic operators in source code. Furthermore, little formations within the line as removal or insertion some words also expand Type-3 code clones.

Figure 7.6 demonstrates clones of Type-4.This category of clones has been classified into meaningful clones, which have different structures, while execute similar estimates. Code clone of Type-4 is generated by rearranging the dependent or independent data statements, alters in the constant or replacement of constants.

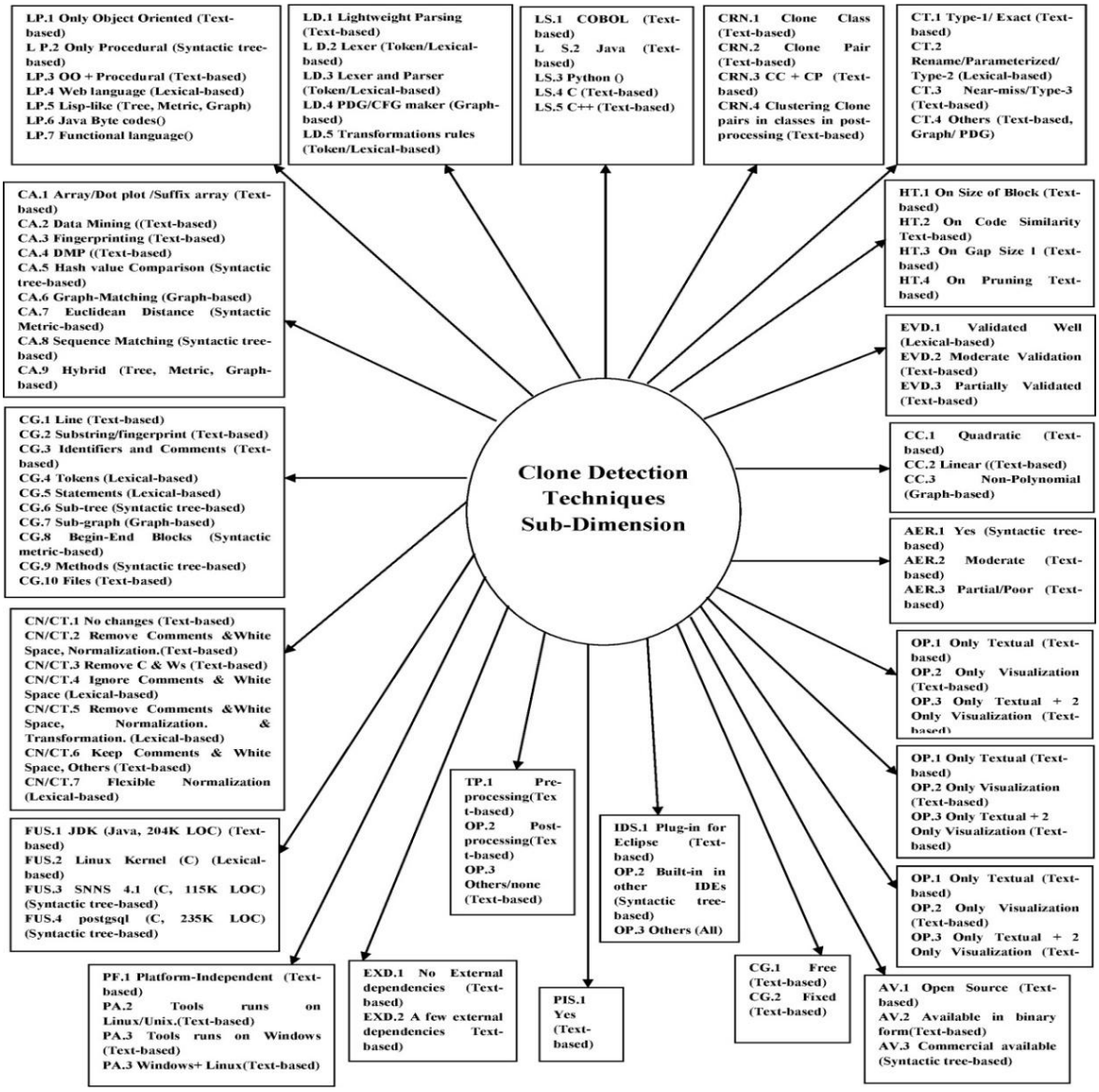



**Figure 7.3:** Clone detection techniques sub-properties

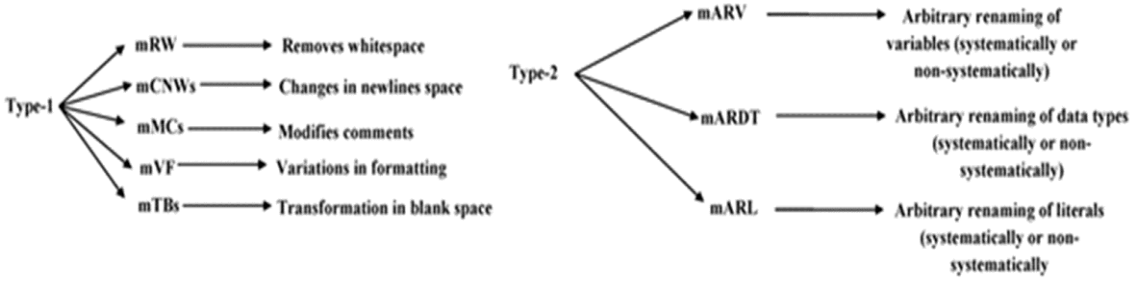



**Figure 7.4:** Editing Taxonomy of Mutation Operators for code clone of Type-1 and Type-2

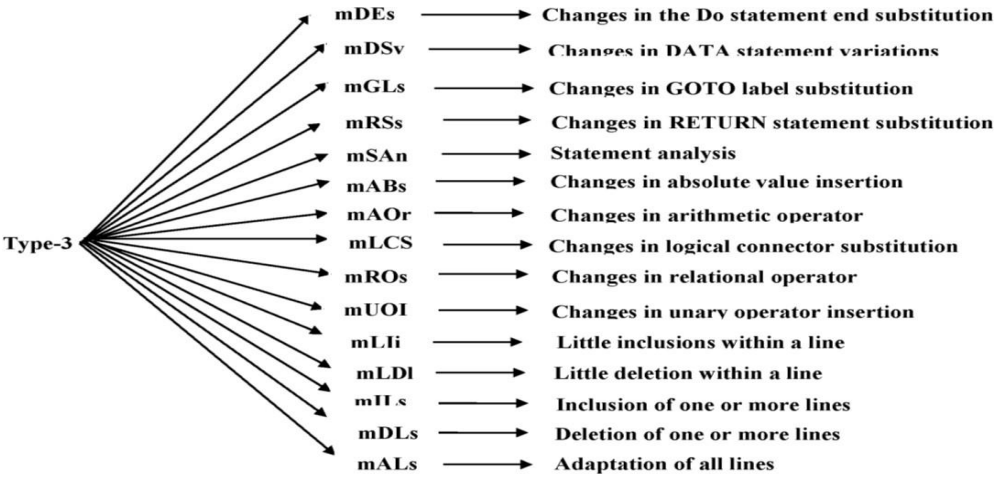



**Figure 7.5:** Editing Taxonomy for Mutation Operators for code clone of Type-3

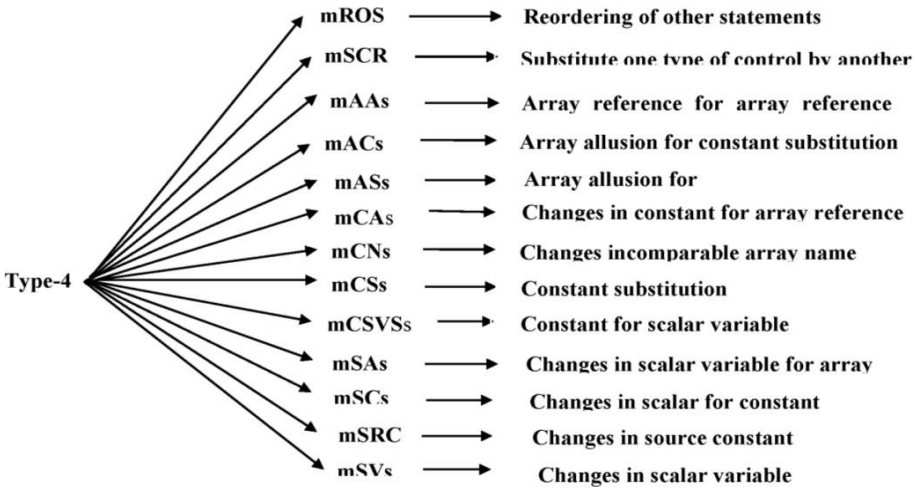



**Figure 7.6:** Editing Taxonomy for Mutation Operators for code clone of Type-4

### 7.2.4 Mutation Operator- Based Scenario for Evaluation and Comparisons

There are numerous detection approaches and tools have been proposed in the literature for code clone. Though, software clone detection approaches are partly estimated in the last few years [152, 164, 165] presented one of insidious abstractions including perceptible assessment of six clone detectors on an extensive Java and C software system. Additionally, in this research inadequate exclusions of the clones were the oracle and several another aspects were selected as possibly influencing the outcomes [68]. The main incompetency of the assessment gets aggravated by the fact that there are no customary criterions for review. This section demonstrates a hypothetical scenario which is based on imitation operator-based editing classification for evaluating clone detection approaches more analytically; we have used proposed mutation operators-based editing method. In the Figure 7.7, we have prepared a miniature set of theoretical scenarios which is idyllic modification in fragment of code by copy / paste. Each scenario satisfies a software clone type as describe in [21] and [148] with surplus

differentiation enhancements. We believe our main objective is to find the exact software clone, indefeasibly generated by the copy / edit source text. Figure 7.7 characterizes software clone scenarios of Type-1 to Type-4.

Type-1 code clones depicted in “Scenario 1”. A programmer replicated a function which subtracts addition and multiplication of a sequence of numbers (1...n).
Scenario S1 consists of five sections as S1 (a) related to modifications in whitespaces, S1 (b) changes in comments, S1(c) refers to the accomplish alterations in formatting, S1(d) inserting blank spaces in code S1(e) adding new line spaces in the code.
Figure 7.7 represent Scenario 2 which demonstrates type-2 clones. To create this scenario code developer creates four replicated copies of source code by some changes in literals, variable names systematically. In Scenario S2 (a) renaming identifiers S2 (b) related to identifiers renaming methodically S2(c) data-types renaming but not systematically, S2 (d) targets replacements of parameters with some expressions.
Type- 3 clones can be generated using Scenario 3 by adding, removing lines in source code, S3 (a) presents little insertion within the sequence S3(b), deletion within line, S3(c) insertion of a new line in this scenario S3 (d) deletion of a line S3(e) modification begins within some line
Figure 7.4 to7.6 shows S4 scenario which is created by software developers using mutation operator-based editing assortment.
Scenario 4 (a) generated by reorders the declaration statements, S4(b) in this phase data independent statements reorder, S4 (c) again reorder data dependent statements S4 (d) replace control statements with “for” or “while”, Scenario 5 exemplifies textually equivalent clone as type- to type-3 and functionally equivalent clone as type-4. Scenario 5 targets on where a code developer ensures hower an approach detects clones. S5 (a) defines syntactic code clones S5(b) how an approach can identify clones in a similar range of code base in a methodical manner. Scenario 6 targets on various forms of code clones, S6 (a) specify type-1 to type-2 code clones while, S6 (b) shows the type-3, type-4 clones.
Figure 7.8 and 7.9 related to the precision and recall of detection tools and methods which is calculated by five scenarios of Figure 7.7, generated by mutation operators-based classification.
Text and token-based tools comparisons shown in Figure 7.8. Roy and Cordy, 2009 [183] presents a scenario which is based on comparisons of detection tools and methods. Further, the authors [183] carried out five scenarios with eighteen comparisons. We have proposed five scenarios with twenty comparisons. Although, a number of scenarios can be generated for evaluating detection tool but it is not feasible to evaluate detection tools and techniques due to

the limitations of detection tools and methods. Figure 7.8 shows the evaluation result of string-based and lexical-based tools as “Johnson” (1994) is text-based tool and it is well defined in scenario 1. Additionally, “Duploc” (1999), is magnificent in scenario 1(a, b) it most likely can identify S 1(c,), in S3 (a, b, e) “Duploc” is additionally G and in S5 (a) “Duploc” is presumably G and scenario S5 (b) it is medium. “Sif” (1994) is M in scenario 1, and it is presumably can identify clone in S2 (c, d), in S3 (a, b) and S4 (a, b). “DuDe” (2005) is great in S1 (a, b) and can recognize presumably in S1(c, d, e) and G in S3 (a, b) and M in S3 (c, d, e). “SDD” (2005) is G in S1 (a, b), S (a, b) and M in S1 (c, d, e), and L in S3 (c, d, e) “Marcus” (2001), M in S l (a, c) S2 (c), S3 (a, b) S4 (a, b, c), and L in S2 (d). “NICAD” (2008), is incredible in S1 and M in S2 it is G in S3 and M in S4 (a, b). “PMD” (2007), is M in S1 (a, b) and most likely can recognize S1 (c, d, e) it is presumably can distinguish clone in S2 and S3 and token-based tool are “Dup” (1995), is great in S 2(a, b) and likely can identify in S1(c, d, e). “CCFinder” (2002), “Gemini” (2002) “RTF” (2007) is W in S1, S2 (a, b) and in S5 (a, b). “CP-Miner” (2006) is G in Si, S2 and low in S2 (d), in S3 (a, b) it is great and medium in S2 (c, d, e) it performs well in S5 ((a, b). “CCFinderX” (2017) is M in S1 and S2.

Figure 7.8 and Figure 7.9 shows the evaluation of clone detection tools. Where G indicates the good, M related to medium, E refers to excellent, while L present the low and at last W shoes the well.

Figure 7.9 shows comparisons of AST (syntactic tree method), Metric (syntactic metric method approach) and graph (semantic methodology). “CloneDr” (1998) is a tree-based tool and it is G (good) in S1, S2 though L (low) in S2(d) or S5(b). Further, it is M (medium) in S 3(a, b). “Asta” (2009) is G in S1 and M in S2 and S5. “Yang” (1991), is G at S1, S 2(a, b, c) while “cpdetector” (2006), E (excellent) in S1, and G in S2 (a, b, c), S5 (a, b). “Deckard” (2007), is additionally G in S1, S2 (a, b, c) and M in S3 (a, b, c, d, e), in S5 it operates considerably. “Tairas” (2006), is G in S1 and M in S2(c), S 5(a). “Clone Detection” (2004), is G in S1, S 2(a, b, c). “Bauhaus” (2002), is G in S1 and M in S2. Metric-based tools are “Konto” (1996), is M in 51, S 2(a, b, c),S 4(a, b, c) however it is L in S 2(d), S 4(d), S 3(a, b, c, d, e), S 5(a, b).Covet (1996) is M in S 1(a), S2, S 3(a, b), S 4(a, b) and it is L in S3 (c, d, e), S 4(c), S 5(a). Davey (1995) M in Si, S2, S 3(a, b), S 4(a, b) and it most likely can't recognize clones in S 3(c, d, e) and it is L in S4 (c). The graph-based tools are “Duplix” (2001), is G in S1, S2 (a) b, c), S4 (an) and M in S2 (c, d), S3, S4 (b). “KomoRag”(2001), G in S1, S4 (a, b), S5 (a) M in S2 (a, b) and L in S2 (c, d), S3(a), S 4(c), and S3 (c) it can identify presumably in S3 (b, d) “GPLAG”(2006) is W(well) in S1 M in S 2(a, b),S 3(c, d ,e), S4 (a, b), L in S2 (c, d), S3 (a, b), S 4(c), S5 (an) It performs G in S4 (a, b). “ConQAT” (2010) is a hybrid approach based tool and it is M in S1, S2 while the proposed

technique (2017) or, in other words a cross breed approach is phenomenal in S1 and M in S2. The itemized depiction is delineated in Table. 7.2, and Table 7.3.

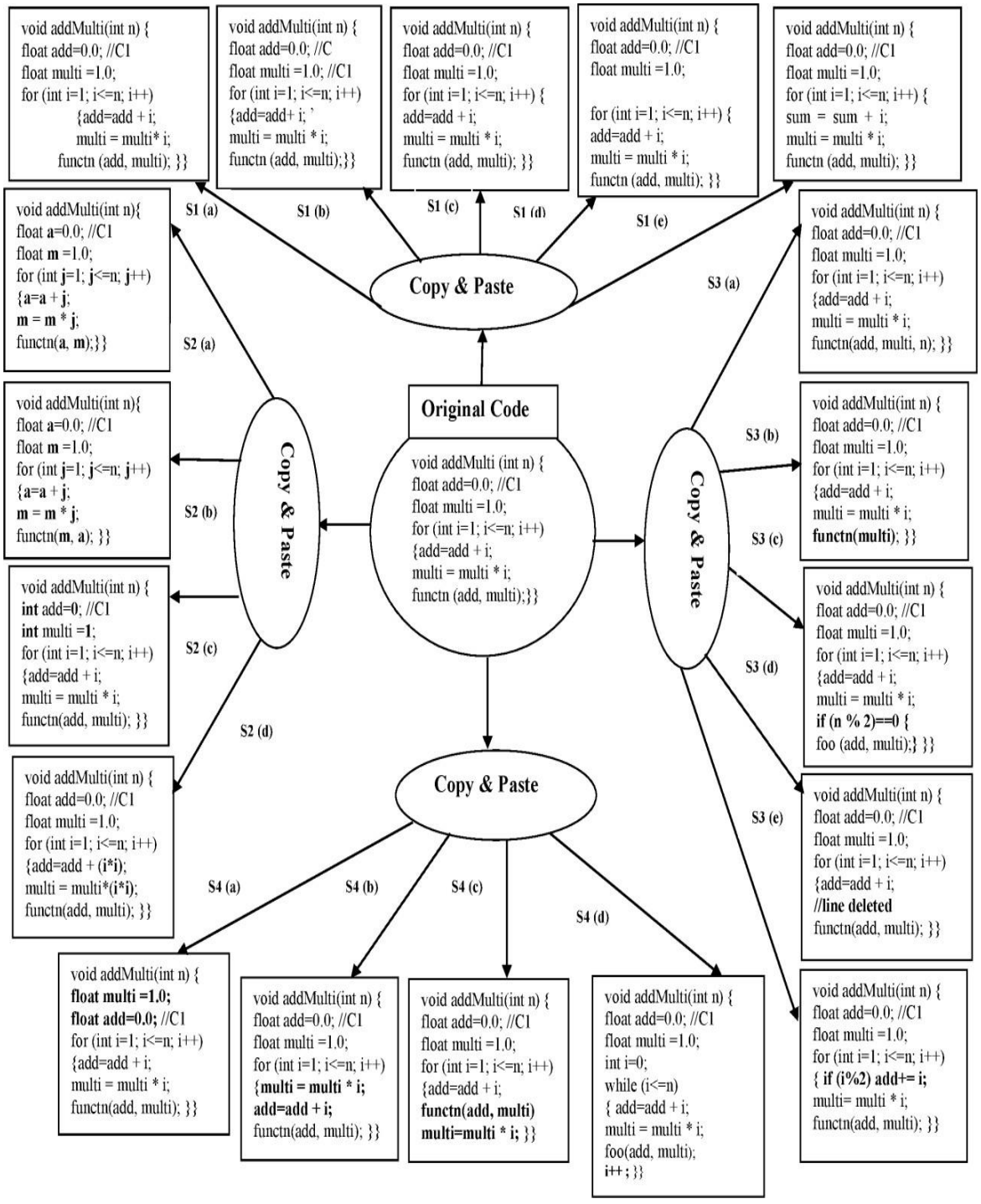



**Figure 7.7:** Editing Taxonomy for Mutation Operators scenarios for distinct software clone

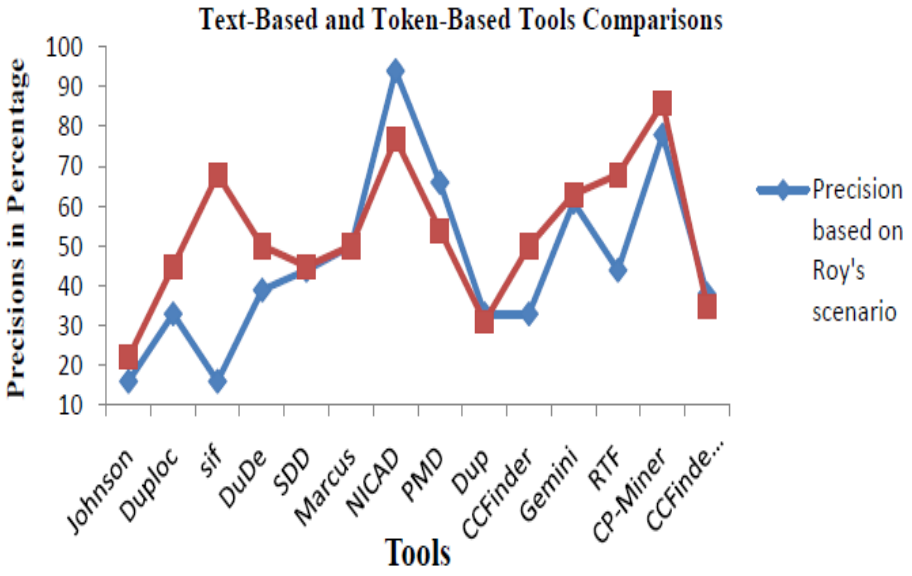



**Figure 7.8:** Scenario-based comparisons of string and lexical-based approaches with exist scenario-based and proposed scenario

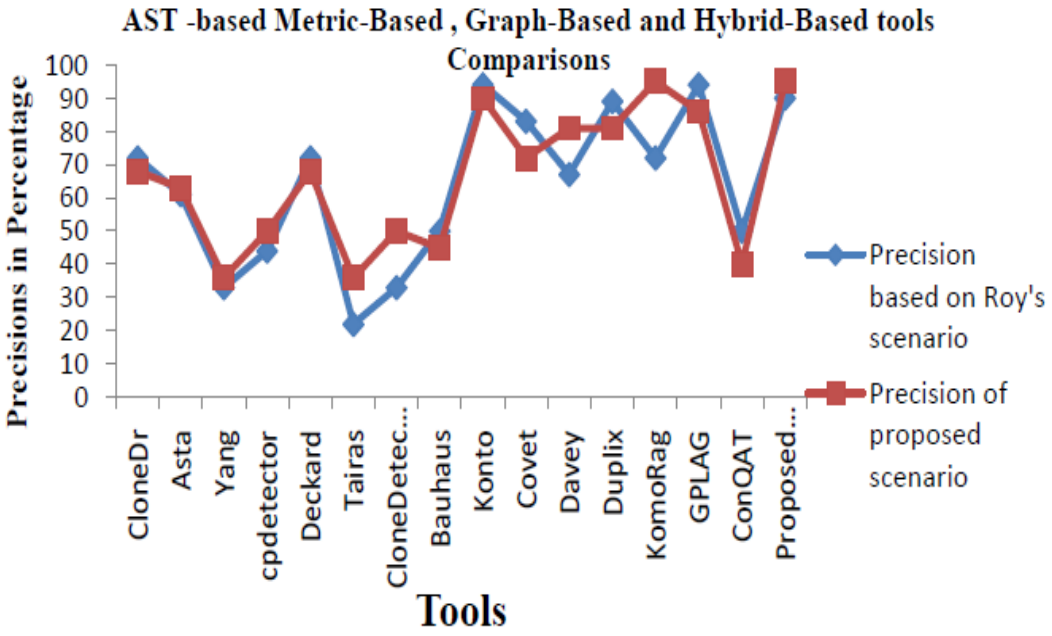



**Figure 7.9:** Scenario-based comparisons of metrics, tree and graph-based approaches with exist scenario-based and proposed scenario

**Table 7.2:** Evaluation of clone detection techniques (string, token and tree) using mutation operator-based scenario

* Excellent # Good $ Medium & Low % Probably can @ Probably !can't

| Techniques | Tools | Scenario 1 | | | | | Scenario 2 | | | | Scenario 3 | | | | | Scenario 4 | | | | Scenario 5 | | Scenario 6 | | Performance Metric |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | a | b | c | d | e | a | b | c | d | a | b | c | d | e | a | b | c | d | a | b | a | b | Precision |
| Text-based | Johnson | # | # | # | & | & | ! | ! | ! | ! | ! | ! | ! | ! | ! | ! | ! | ! | ! | ! | ! | ! | ! | 22 |
| | Duploc | * | * | ! | & | & | ! | ! | ! | ! | # | # | ! | ! | # | ! | ! | ! | ! | % | $ | $ | ! | 45 |
| | Sif | $ | $ | $ | $ | & | ! | ! | & | & | % | % | ! | ! | ! | % | % | % | ! | ! | ! | $ | ! | 59 |
| | DuDe | * | * | & | & | % | ! | ! | ! | ! | # | # | $ | $ | $ | ! | ! | ! | ! | ! | ! | $ | ! | 50 |
| | SDD | # | # | $ | $ | & | ! | ! | ! | ! | ! | # | & | & | & | ! | ! | ! | ! | ! | ! | & | ! | 45 |
| | Marcus | $ | ! | $ | % | % | ! | ! | $ | & | & | & | ! | ! | ! | & | ! | $ | ! | ! | ! | & | ! | 50 |
| | NICAD | * | * | # | & | & | & | $ | $ | $ | & | & | & | & | & | & | ! | ! | ! | ! | ! | # | & | 77 |
| | PMD | * | # | $ | & | & | $ | ! | ! | $ | ! | ! | & | ! | ! | ! | ! | ! | ! | ! | & | & | # | & | 54 |
| Token-based | Dup | * | * | ! | ! | ! | * | ! | $ | ! | ! | ! | ! | ! | ! | ! | ! | ! | | ! | $ | $ | $ | 31 |
| | CCFinder | $ | $ | $ | $ | $ | $ | $ | $ | $ | ! | ! | ! | ! | ! | ! | ! | ! | ! | ! | | $ | @ | 50 |
| | Gemini | $ | $ | $ | $ | $ | $ | $ | $ | ! | $ | $ | $ | $ | $ | | ! | ! | ! | ! | ! | ! | @ | 63 |
| | RTF | # | # | # | $ | % | # | # | # | % | % | % | ! | ! | ! | ! | ! | ! | # | # | ! | # | % | 68 |
| | CP-Miner | # | # | # | $ | % | # | # | # | & | # | # | $ | $ | $ | ! | ! | ! | # | # | # | # | % | 86 |
| Tree-based | CloneDr | # | # | # | ! | ! | # | # | $ | & | $ | $ | ! | ! | & | & | ! | ! | ! | & | $ | $ | % | 68 |
| | Asta | # | # | # | $ | % | $ | $ | $ | $ | $ | $ | ! | ! | ! | | ! | ! | ! | $ | $ | $ | ! | 63 |
| | Yang | # | # | # | % | % | # | # | # | ! | ! | ! | ! | ! | ! | 1 | ! | ! | ! | ! | ! | ! | ! | 36 |
| | cpdetector | * | * | * | $ | % | # | # | # | ! | ! | ! | ! | ! | ! | ! | ! | 1 | ! | # | # | # | ! | 50 |
| | Deckard | # | # | # | $ | ! | # | # | # | ! | $ | $ | $ | $ | $ | ! | ! | ! | ! | # | # | # | ! | 68 |
| | Tairas | # | # | # | $ | ! | ! | ! | $ | % | ! | ! | ! | ! | ! | ! | ! | ! | ! | $ | ! | $ | ! | 36 |
| | CloneDetection | # | # | # | $ | % | # | # | # | $ | ! | ! | ! | ! | ! | ! | ! | 1 | ! | @ | ! | $ | ! | 50 |

**Table 7. 3:** Evaluation of clone detection techniques (syntactic and semantic using mutation operator-based scenario

* Excellent # Good $ Medium & Low % Probably can @ Probably !can't

| Techniques | Tools | Scenario 1 | | | | | Scenario 2 | | | | Scenario 3 | | | | | Scenario 4 | | | | Scenario 5 | | Scenario 6 | | Performance Metric |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | a | b | c | d | e | a | b | c | d | a | b | c | d | e | a | b | c | d | a | b | a | b | Precision |
| Metric-based(Syntactic-approach) | Konto | $ | $ | $ | & | % | $ | $ | $ | & | & | & | & | & | & | $ | $ | $ | & | & | ! | $ | ! | 90 |
| | Covet | $ | ! | $ | ! | ! | $ | $ | $ | $ | $ | $ | & | & | $ | $ | $ | & | ! | & | ! | % | ! | 72 |
| | Davey | $ | $ | $ | & | ! | $ | $ | $ | $ | $ | $ | % | % | % | $ | ! | & | ! | & | ! | $ | ! | 81 |
| Graph-based(Semantic approach) | Duplix | # | # | # | & | & | # | # | # | $ | $ | $ | $ | $ | $ | # | & | ! | ! | $ | ! | $ | ! | 81 |
| | KomoRag | # | # | # | & | & | # | $ | $ | & | & | @ | & | @ | @ | $ | $ | & | & | $ | ! | $ | & | 95 |
| | GPLAG | # | # | # | % | $ | # | ! | $ | & | & | & | $ | $ | $ | # | # | & | $ | ! | ! | $ | $ | 86 |
| Hybrid approach | ConQAT | $ | $ | & | & | ! | & | & | ! | ! | & | ! | ! | ! | ! | ! | ! | ! | & | ! | ! | $ | ! | 40 |
| | Proposed approach | * | * | * | * | * | # | $ | $ | $ | & | & | & | & | & | % | % | % | % | # | & | * | ! | 90 |

### 7.2.5 Conclusion and Future Work

This section presents performance evaluation of code clone detection tools and techniques in two different perspectives. First, to contrast tools and methods by code clone property well as sub-property. Subsequently, to assessed clone detection tools by mutation- operator based created the clone classification. Moreover, classification based on Mutation-Operator was accustomed to generate a scenario to evaluate the tools and methods for clone detection. However, Roy et al. [184] employed these scenarios for assessing tools and methods for clone detection, the barriers to their technique were that they evaluated using only five scenarios and not using mutation operators for creating clone. Further, to applied mutation operators for generating distinct code clones types, and by that, we induced "six scenarios, and by using these theoretical scenarios, we have extent how well the various clone detection techniques may perform, based on their detailed facets as well as sub-facets. However, this is not a real evaluation;" Instead, it shows the wider depiction of the interior of each identity of the clone approach in executing duplicate code for each scenario. Estimated diligence has some limitations, such as ROY et al [21] used only five scenarios, due to the lack of tools access and many other comprehensions. Apart from it for the future, we incorporate clone identity approaches and complete evaluation of equipment using grand scenarios.

## 7.3 MUTATION OPERATOR-BASED AN AUTOMATIC FRAMEWORK FOR INJECTING AND DETECTING DUPLICATED CODE

### 7.3.1 Introduction

The experimental assessment of testing methods plays a significant role in code testing like-wise as in code clone detection analysis. One common practice is inserting flaws, either manually or by employing mutation operators [166]. Generally, experiential valuations are accustomed induce two or a lot of techniques, that are exclusionary for acceding with some detection relevant activity. Testing experimentation sometimes encompass a collection of subject programs with known flaws or errors. Moreover, in code clone detection In addition to evaluating code cloning detection methods, some incorrect version or duplicate code is required. Many scientists have taken several methods to present a bug in the program to create a defective version; bugs are Infiltrating by hand or automatically generated in program text.

Typically, we analysis a consequently-created variant as a result of applying some editing activities within the program text. Although, these editing activities are compiled consistently with well-defined rules that are referred to as mutant operators, and the consequential faults versions are called mutation generation [169]. The mutation operators similarly accustomed create probable bugs in code clones to alter the program [184-185]. Copying a program text from one section to another section by copying and pasting is a frequent process in code development. As a consequence, identical copied code segments are known to as code clones, and therefore the entire procedures are called code cloning [184]. Previous, survey reported that a considerable section (20-59) of code within software is duplicated code [15, 17, 18, 19, 56, 111]. A flaw single out in one section of code, and so all sections of program text ought to be checked for similar flaw [61].

The duplicated code may significantly enhance the efforts when increasing source code [17]. Although, varied code clone detection methods are exist in the literature, and there are several comparisons and valuation notions relating them in various perspectives [52, 152, 163-165]. Nevertheless, it is challenging to evaluate several clone detection tools, due to code clone detection approaches having specific attributes, hence these studies illustrates considerable amplifications to the code clone detection research era [68]. Typically, short assessment is irritated as there aren't any universal analysis benchmarks. It’s thorny to search out such a general norm as every technique has its some options and is meant for distinctive reasons.

Typically, too little assessment is agitated as there are not any universal analysis assessment benchmarks. It's thorny to seek out such a general norm as every technique has its own some attributes and is intended for distinctive inductions.

This section presents an outline of mutation testing-based analysis framework. The purpose of the proposed method is to layout of an evaluation structure based on mutation, used to evaluate tools and techniques for clone identification. The proposed structure remains under implementation, appeared to measure and increase the tools for detection of clone. This code will begin with the concept of cloning and mutation testing with the introduction, from which an editing nomenclature of mutant operators may be derived. Besides, by these operators of mutation, we give an emphasis on framework to evaluate the tools for detection of code clone.

### 7. 3. 2 Overview of Testing of Mutation

A type of white box test, which is employed for unit testing, is called Mutation. In the mutation test, some statements of the source code were changed (mutated) and tested on test-cases to verify that cases of test were able to find errors. The change is kept very small during the mutant program so that it does not modify the motive of the program. The concept of testing using mutation is to make a sentence constructive change in the code, to clearly create a defective version (as a mutant) by clearly defined rules (mutant operators) [14]. The imitation testing first time emerged within the 1970 by [13], during class term paper. The first research article published by the authors [167, 168,186]. The author [170] accomplished a review on the subarea of mutation testing that was a frail assessment whereas the condensed mutation was presented by [187, 188]. The researchers [18 9, 190] provided an initial chapter on mutation testing. Subsequently, an impressive survey on [28] mutation testing was published, which provided an overview of the existing optimization techniques for imitation testing. After that [191] outline, "Mutation testing, as a fault-based test, presents a test standard called Mutation Competency Score". To identify "the defects, the efficiency of the test can be calculated using this score related to its facility." The current assessment was carried out by the authors [192]. They provided a scientific literature review on the observation of the application of the mutation test. The methods of mutation testing and mutant operators are categorized into three types and two type's classes, reactively, which are demonstrated in Figure 7.10.

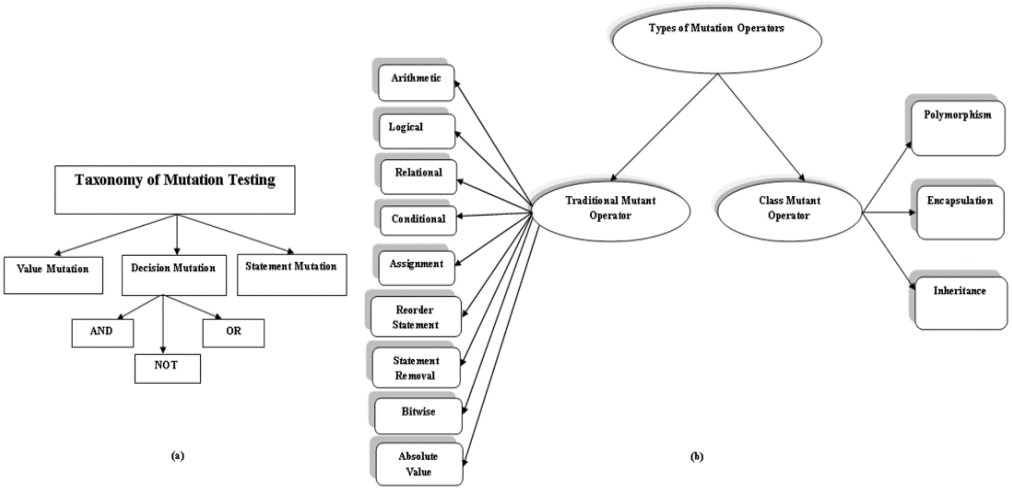



**Figure 7.10:** Taxonomy for the (a) Mutation testing, (b) Mutation operators

Figure 7.10 (a) presents the taxonomy of techniques based on mutation testing as 1) the key constant's value and types of mutation operators changes in the value mutation testing. 2) Decision Mutation testing, optimization within the control details of the program, or not, 3) the details of the source code recognized in the Statement Mutation Testing Techniques. Figure 7.10 (b) illustrates various forms of mutant operators. There are two types of mutation operators namely as traditional mutation operator and class mutants operators. Further, the mutation operators again assorted into three types as arithmetic, logical, relational, conditional, etc. 2) Class mutant" operators can be in the form of heritage (eliminate a latent variable), allotropy (same name different works methods) and encapsulation (insert, remove, and transmutes variables to make mutations).

### 7. 3. 3 Related Work

The basic idea behind mutation testing is creating exactly similar variants from the original program. Each mutant contains at least one artificial modification [193]. There is no mutation testing-based estimation outline given within the literature; but, there are some experimentations that distinction and valuate code clone detection techniques and tools. Thus, this section presents a blueprint of the existing tools evaluation analysis from the studies. The Primary research [164] was completed on "three state-of-the-art work on clone detection tools" to assess the two plagiarism theft identification tools and their performance. The main factor that they did was validating the whole duplicate code clone applying all the techniques for his experiment. Several methods evaluated against various constraints by human oracle that successively uses many metrics that various measure answerable for measure totally different aspects of the identified code clones. Though they efficiently identified all the

candidate clones, the primary constraints of this case study are regarding size and system quality. The primary objective of their research was to help in a preventive maintenance task which had leverage invalidating the candidate clone. Authors have [152] accomplished tool comparison which was invented after considering the limitations of [164] experiment. Further, the authors [152] conducted this experiment with similar three software clone detection tools used [164] in their analysis and they also used three additional clone detection tools in his experiments. The system software system [164] had used a various set, accounting to four Java and four C. Moreover, once the authors [164] studies were taken into cogitation whereas legitimize the candidate clones, a human oracle was then used for identical. Although, being the thorough study so far, the clone candidate being oracled was little fraction considering that alternative factors might need pretentious the results [68]. Although, authors [52] extensive this research with an ideal application of many tools, however while not addressing something to subdue the obstructions of Bellon et al., research. Evaluation of three envoys for detection techniques of code cloning by [165], provided comparative fallout, type of duplicates, scalability, some false experiment matches and ineffective matches. Nevertheless, limited size cases and prototype implementation were used by them rather than authentication tools. They wanted to draw the conclusions of the identification technique for a humble act as the replacement of the conceptual analysis of the identity approach. The researchers [163] reported an enthralling study; they evaluated various clone detection techniques in respect of identifying crosscutting suspicions.

Recently, various researchers proposed the number of techniques for mutation testing. Author [194] proposed an efficient mutation testing framework for multi-threaded code which can reduce the time required for mutation testing of multi-threaded code. They introduced a tool named MuTMuT which is based on four optimizations and one heuristic method used mutation testing in a safety-critical industry Using C and ADA's high integrity subset [195]. They recognized mutant types adequately and analyzed the main reasons for failure in test cases.
In addition, they also provided practical assessment of mutation testing application in the Airborne software system. One of the main issues regarding mutation testing was high cost, due to the creation of mutants, execution of mutants and calculation of their scores. Mateo and Usaola [196] proposed a mutant schema with extra code (MUSIC) which reduces the mutation cost through uncovered mutants. Though, this method defines the proclamations enclosed by the tests in the original system, to out the mutant implementation, because tests are executed only against the mutants whose mutated statement is enclosed by tests. Authors [197] measured the

complexity of mutants and prioritized them by how easy or hard to manifest them. The mutation testing presented in perspective of python program [198]]. They showed how mutation testing could be adequately handled in the python environment. A mutation reduction method proposed regarding program structure [199]. Although, they used two path-aware heuristic rules named loop-depth and module-depth rules and combined them with operator-based selection and statements to develop four mutant reduction approaches. The researchers [200] evaluated mutation at the class level while existing method analyzes mutation at the traditional standard. Further, they proposed a MuCPP system which is based on the class mutation operators of the C++ programming language. An improved genetic algorithm presented [201] which is a search-based approach to diminish the statistical expense of transformation testing. However, they compared their tool eMuJava, which include state-based and control oriented fitness functions, with others accepted fitness functions. However, the aforementioned mutation-based testing approaches have been presented in the literature. All these approaches were focused on mutation testing, while mutation word first time was used [184-185] in clone detection research field in a brief way; they proposed identification tools for clone using mutation / injection-based framework, which provide an editing taxonomy for different generation of clone. As per the authors' best knowledge, there is no thorough work on mutation testing operators in perspectives of software clone detection tools. There is no standard benchmark available for evaluating clone detection, and there is no empirical evaluation, which explicitly determines the utilization of mutants in the clone detection research field. The author contribution in this manuscript is summarized as follows:

- To use the mutation testing concept in the clone detection research area.
- Uses of mutation operators for generating various types of software clones.
- Proposed an evaluation framework using mutated code clones.
- Evaluation clone detection tools by mutated code clones.

### 7.3.4 Proposed Evaluation Framework

This section presents details of the proposed structure which is based on mutation testing as shown in Figure 7.13. It starts with the essential components of the proposed framework and then thrashes out clones terminology, precision, and recall of the tools. Although the introduced structure is based on mutation testing and it acts by the principle of mutation testing. Figure 7.13 illustrates the conceptual layout of the proposed framework. The proposed structure can be divided into two key types. Firstly, Clone Generation Phase, in this phase, software clones is

generated from the original code with the help of mutation operator-based editing taxonomy. The second phase, Tools Evaluation Phase, in which mutated systems is used to assessment the performance of detection tools for code clone. The detailed discussions of the proposed framework are shown below.

### A. Clone generation phase

- Input Original Code Base: At the initial stage of the framework, we input the target code base, which is shown in Figure 7.11. To find out that tool would be important for such a valuation in terms of recall and precision.

Figure 7.11 shows an example of a code base which is taken as input in the proposed framework. The original code retrieved from an open source project named as wet lab [122] which is an open source project of c++ language.

```
//Date: Oct 4 Start :5:23PM End :5:30PM
//Find The Pair With Given Sum
\\Testing Comment Inserted//Complexity : O(n)
#include<iostream>
#include<vector>
using namespace std;
int main()
{
vector<int> v = {1,2,4,5,6,7};
int sum ,left =0 ,right=v.size()-1;
cin>>sum;
while(left<right)

{
  if(v[left]+v[right]==sum)
  break;
  else if(v[left]+v[right]>sum)
  right--;
  else
  left++;
}
if(left<right)
cout<<"Pair Having Sum  "<<sum <<" found at location "<<left+1<<" and "<<right+1<<" value
s are "<<v[left]<<" and "<<v[right];
```

**Figure 7.11:** An example of original code base

```
//Date: Oct 4 Start :5:23PM End :5:30PM
//Find The Pair With Given Sum
//Complexity : O(n)
#include<iostream>
#include<vector>
using namespace std;
int main()
{
vector<int> v = {1,2,4,5,6,7};
int sum ,left =0 ,right=v.size()-1;
cin>>sum;
while(left<right)
{
  if(v[left]+v[right]==sum)
  break;
  else if(v[left]+v[right]>sum)
  right--;
  else
  left++;
}
if(left<right)
cout<<"Pair Having Sum  "<<sum <<" found at location "<<left+1<<" and "<<right+1<<" value
s are "<<v[left]<<" and "<<v[right];
else
```

**Figure 7.12:** An illustration of code clone (type-1)

Figure 7.12 depicted an illustration of duplicated code which is generated using mutation operators based editing taxonomy. Although, it is generated using source code.

- **Selection of Code Segment Arbitrary**

Once selecting the code, an ideal partial number of the existing source code segment is chosen

either automatically or arbitrarily with the code base for the clone mutation.

- **Stored in Source Database**

Randomly chosen code fragments are accumulated in the source database, and then mutants will be created by using these randomly selected code fragments.

- **Create mutants of random code segment**

The classification based on edit of mutation operators used for creating mutant-versions of arbitrarily elected code fragments, which are hold on within the mutant source database. An illustration of mutant-variant of the prime code base is shown in Figure 7.13. Several mutants may be created for one code section. The mutated adaption of the code base after substitution is going to be sustaining as input to clone detection tool. The detection tools are going to be evaluated by what number clones it'll identify precisely and how quick they identify clones.

**A. Tools evaluation phase**

In this phase, each of the mutated code is stored in the database, and then each mutated code is fed into clone detection tools as an input for assessing and analyzing clone detection tools. The main key feature of this phase is the threshold. We defined a threshold for clone detection tools. If a tool does not fulfill the threshold limit, then that will be eradicated, and a new tool will be selected for evaluation.

- **Substitute arbitrary code fraction with mutants**

The transformed code has been developed for every form of mutant code for the program text. The initial code base was restored with indiscriminate code segments inside the generated mutated original code base.

- **Select a clone detection tool**

In this phase a tool will be selected and then mutated code based is given as input into the selected tool.

- **Run injected code clones on tool**

The input for the identification tools after the replacement will be code based mutated variants. The detection tool will be assessed by how many clones detected by detection tools accurately and rapidly.

- **Evaluation report of tool**

The assessment report decides for the tools. It analyzes the performance of the tools based on accuracy (not precisely detected clone), recall (total clones), scalability (supports large databases) and portability (language support).

- **Define threshold**

The primary purpose of this step is to decrease the execution time as well as the price of method. A threshold outlined by a user for a clone detection tool and also the tool crossing the limit are going to be eliminated, and it'll choose a replacement tool for analysis. Apart from this, the tool which has less precision and recall is going to be surplus, and high precision-recalling tools are being evaluated. This will be helpful to those users who are demanding tools with high precision-recall.

- **Eradicate iinefficient tool**

Most tools have returned high precision and low recall and due to this reason they are have less accuracy. Thus, it's induced to the notice that may be a. step of valuated the outcomes of tools that ought to be saved at the end.

Statistical Analysis Report

Once the experimentation is accomplished then the evaluation database is used to calculate clone detection tools accuracy as precision, recall, portability and scalability for each type of clone.

### 7.3. 5 Clone Terminology

Clone detection tool returns clones as "clone pairs (CP), clone classes (CC)" or both. The relationship among clone fragments represent by these two terms. In addition, the clone equality relationship among the code segments is an affinity relationship that is described in [59]. Clone relationship exists between two parts if they are structurally or semantically. A relationship exist among code pair of code fragments then it's called clone pair, while the clone class is a union of all clone pairs [21].

**Definition 1: Code Segment**

A code segment (CS) consists of any subsequence of code string. Any granularity defines it as fixed (predefined syntactic-boundary as a function, begin-end block) or free (no syntactic boundary). Granularity detects a CS in the original program, and it is implied as (CS.Filename, CS.BeginLine, CS. End Line).

Let P = {0, 1, 2…..) and P+= {1,2….). For p € P, denoted by O (p) A the set of n operations on A and set OA: = Up€P+ O(p)A. A subset CS⊆OA.

**Definition 2: Software Clone**

A code section CS2 is a copied fraction clone of another source portion CS1 in the event that they are indistinguishable by some given meaning of similitude that is f (CS1)= CS2, where f is

a likeness function(textual or semantic). Further, when two code sections are like one another, they are called clone sets.

CP = (CS1, CS2)

**Definition 3: Software Clone Types**

The attributes can classify code clones as textual similarity-based or syntactic-based, and the other is functional similarity-based or semantic-based [111].

**Definition 4: Code Segment Encompassment**

If two or more code segment is contained within same file or the boundary of line numbers of CS1 is within the boundary of line numbers of CS2 in code form. File contained (CS CS1, CS CS2)

If ((CS1.FileName==CS2.FileName) && (CS1.BeginLine>=CS2.BeginLine) && (CS1EndLine<=CS2.EndLine))

### 7. 3. 6. Measuring Recall

The primary aim of the proposed framework is to automatically-inject software clones in the source code which are generated by using mutation-operators.

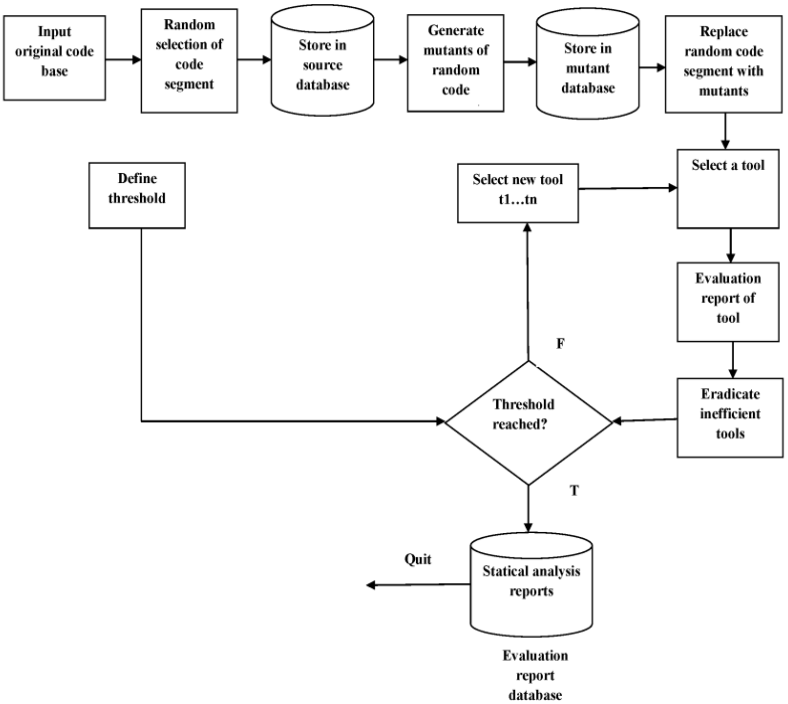



**Figure 7.13:** The proposed mutation-testing-based automatic evaluation framework (a) Clone Generation Phase (b) Clone Tools Evaluation Phase

Further, we evaluate the clone detection tool's accuracy regarding precision and recall. The proposed framework addresses recall for each type of software clones, and for each kind of

mutation operators for all the tools. In this proposed framework, the detectable clones are our injected mutant clone versions of the source code, and that will be inserted into mutated source code base, randomly or automatically. The mutation testing-based techniques make recall simpler. Moreover, if the mutated code segment moCS of original code segment oCS inserted into the mutated code base mioCB of source code base oCB is "killed" (oCScs, moCS) is detected by clone detector and their threshold level value which returns 1 if detector's minimum threshold value is greater than 1 and maximum value less than 100, otherwise it will return to 0. The main objective of the threshold value is to minimize the execution time and complexity. The threshold value depends on the user's requirements, and the user can define threshold between 1 to 100 because recall cannot be less than 1 and cannot be higher than 100.Recall = (Number of detected clones*100)/ Total number of clones

RT (oCS, moCS) = {return 1, if (oCS, moCS) is detected by tool T in mioCB->THLV (minimum value>=1 & maximum value <=100) otherwise 0.

Where THLV means threshold level value. The similar code segments can be inserted or injected randomly, any number of times, in the original code base oCB and generate “different mutated variants of oCB as mioCB I, mioCB2....mioCBn. The proposed framework used mutation operators for creating various types of mutated versions of source”code and then inserting them randomly several times to check the sensitivity of the clone detector. However, the random segment selector selects m code base and mutation operator’s dmOP will mutate each of them for generating mutated versions of code segments as“moCS1, moCS2....moCSm. Hence, the recall for mutation operator dmOP for clone detector is given as follows.

RTdmOP= $\sum_{i=1} n * mRT(oCSi) / n * m$

The type-1 software clones used four types of mutation operators(mCW, mCNWs, mCC, mCF, mCB) and their combination and their (mCW+mCNWs), (mCC+mCF), (mCF+mCB),(mCW+mCC), (mCW+mCF), (mCW+mCB), (mCNW+mCC),( mCNWs+ mCF),( mCNW+mCB), (mCC+mCB), and mCF+mCB) can be applied to the m code segments(if we allow operator repetition, then a number of combination can be generated) and each of which is inserted n times into the code base. Therefore, the recall of clone detector tool T for type-1 can be defined as:”

RTdmOP= $\sum_{i=1} n * m * (5+11)$ RT $(oCSi, moCSi) / m * n * (5+11)$ = {return H, if (n, m) is detected by clone detector tool T in mioCBTHLV (minimum value>=1 & maximum value <=100) otherwise L.

Where H means high recall L means low recall and5 indicates the number of operators and 11 indicates the number of combinations.

The overall recall of clone detectors can be defined as:

RTdmOP = $\sum_{i=1} n*m*(S+C)$ RT $(oCSi, moCSi)/m*n*(S+c)$ = {return H, if (n, m) is detected by tool T in mioCB -> THLV (minimum value>=1 & maximum value <=100) otherwise L.

Where H means high recall L means low recall and S indicates the number clone mutation operators and C indicates the number of combinations.

### 7.3.7 Measuring Precision

Precision measures unnecessary items which appeared in the results. Preferably, accuracy (precision) should be high when recall increases, but practically it is difficult to accomplish. The precision definition is shown below. Precision = "(Number of correctly detected clones* 100)/ Total number of detected clones."

The precision of a tool can be calculated as a mutated code segment moCS generated by using mutation operators dmOP, and clone detector tool T returning k clone pairs,(moCS,CS1),(moCS, CS2)…(moCS, CSK) in mutated code base mioCB.

PTdmOP= w.r.t. t. single insertion of moCS = a/k {return 1, if (oCS, moCS) is detected by tool T in mioCB-> THLV (minimum value>=1 & maximum value <=100) otherwise 0.

here a means accurate detection and k means number of clone pairs returned by a clone detector T.

The overall precision of the clone detector tool regarding the number of mutation operators and number of combinations which is applied n times to m code segments is shown below.

PTdmOP = $\sum_{i=1} n*m*(S+C)$ / $\sum_{i=1} n*m*(S+C)$ = {return H, if (n, m) is detected by clone detector tool T in mioCBTHLV (minimum value>=1 & maximum value <=100) otherwise L. Where H means high precision and L means low precision.

### 7.3. 8. Conclusion and Future Work

Assessment of tools and techniques for detection of code cloning is an emerging research field in today's scenario. Previous experiments had different obstacles to evaluate tools and techniques for clone detection and therefore, a motivational cannot present comparative survey. Thus, in this section, a specific mutation test-based evaluation framework is discussed to assess

the tools for clone detection used by testing the community in the past thirty years. Moreover, this section encompasses operators for mutation testing, and provides insight into concerned task of mutation testing. Although, the proposed framework under the implementation phase till now; but it is secured that the proposed layout will be helpful for truthful analogous result for different tools, in fording deliberately generated code clones. Further, the anticipated framework can injected thousands of mutated variants is injected into the huge codebase system which is created with the help of mutation operator's assortment. Moreover, code clones detection tools will be assessed by how many copied code clones they can distinguish which is embedded in the code base. In future, we experimentally assess a transformation testing-based proposed structure for assessing clone recognition apparatuses.

# CHAPTER 8
# CONCLUSION AND FUTURE WORK

The obfuscation is a security mechanism against reverse engineering, thus obfuscation's security is ensured at three different aspects of reverse engineering. The code clone is used to provide security to obfuscation against static and dynamic analysis attack in the proposed approach. Although, code clone used to protect code from the reverse engineering attack but on the other side code clone having a negative side too as it increases maintenance cost, execution speed, the strain on system resources, bad impact on the design, and bug propagation [20]. Moreover, the software clones have adverse effect in software evolution and maintenance because if a bug has been perceived in one segment of source code, then it requires corrections in the entire source code. In response, a few decades ago some code clone detection tools and techniques have been proposed [20]. The recent survey [21] reported that none of the tools and techniques had been introduced which can identify duplicated code with exalted accuracy. Therefore, there is a need to develop an approach which can detect code clones from the source code with high, "accuracy portability, scalability, and robustness." Thus, the proposed approach detects type-1 software clones [20] by a hybrid approach which has been discussed in chapter 5. The proposed method is based on lexical perception in which complete program is partitioned into tokens", and it detects type-1 software clones (generated using mutation operators) with high precision and recalls and compares it with the existing method. By using a directed acyclic graph (DAG), the type-2 software clones have been detected. "The DAG is a compiler optimization technique which is used to removes common sub-expression from the source code. Further, non-trivial (functionally identical) code clone can be detected by the program dependency graph (PDG), and trivial software clones have been identified by using "control flow graph (CFG) and reduced flow graph (RFG)". Furthermore, the experimental valuation of software clone detection tools and techniques plays an imperative role in software testing and software clone detection research. Thus, there is need to evaluate the performance of existing code clone detection tools in teens of software metrics as precision, recall, portability, scalability, and robustness. Hence, the proposed approach evaluates software clone detection tools and techniques from two perspectives. Foremost, code exposure tools and approaches have been evaluated by the recall, portability, and robustness subsequently, they evaluated by mutation

testing-based generated test-cases which have been discussed in chapter 7. Moreover, it is not mentioned in the literature how to generate code clones in the source code, and how to inject code clones in the source code. Thus, it is essential to propose a generic method for code clone creation and proposed an automatic framework for code clone injection. Thus, the proposed approach is mentioned in chapter 7. It uses mutation operators for generating variants of software clones by using an editing taxonomy which is based on mutation operators. The mutation-testing based scenario evaluates software clone detection tools and techniques by effectiveness criteria (precision, recall). Thus, there is a need to develop an approach which can detect trivial software clones (structurally similar) and non-trivial software clones (functionally identical). The software clones can be injected by hand or by using mutation operators [166]. However, the mutation operator generates software clones by editing activities, and several clone detection tools have been experimentally evaluated it the past. A lot of efforts have been done for empirically evaluating and analyzing various abreast. The current survey exhibits that various attributes "that could use the genuineness of the result of such assessment have been foreseen because of the absence of legitimized code clone benchmark Thus, to overcome this mutation testing-based automatic evaluation structure has been proposed for evaluating software clone detection tools and techniques.

The proposed structure injects software clones automatically in the source code. However, the proposed structure is not completely implemented so in future it will be reported. In future, we will implement the proposed framework which can automatically inject software clones (generated using mutation operator-based taxonomy) as well as it can identify duplicated code automatically from type-I to type-4 software clones.

# REFERENCES


[1] "Annual bsa global software piracy study," International Planning and Research Corporation, Tech. Rep., 2016. [Online]. Available: http://globalstudy.bsa.org/2016/downloads/studies/BSA_GSS_InBrief_US.pdf

[2] J. F. Gantz, A. Florean, R. Lee, V. Lim, B. Sikdar, S. K. S. Lakshmi, L. Madhavan, and M. Nagappan, "The link between pirated software and cybersecurity breaches," *IDC White Paper Google Scholar*, 2014.

[3] M. Fossi, G. Egan, K. Haley, E. Johnson, T. Mack, T. Adams, J. Blackbird, M. K. Low, D. Mazurek, D. McKinney *et al.*, "Symantec internet security threat report trends for 2010," *Volume XVI*, 2011.

[4] C. S. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation-tools for software protection," *IEEE Transactions on software engineering*, vol. 28, no. 8, pp. 735–746, 2002.

[5] J. Nagra and C. Collberg, *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Pearson Education, 2009.

[6] J. Cappaert, "Code obfuscation techniques for software protection," *Katholieke Universiteit Leuven*, pp. 1–112, 2012.

[7] C. Collberg, C. Thomborson, D. Low, and D. Low, "A taxonomy of obfuscating transformations. department of computer sciences, the university of auckland," Technical Report 148, July, Tech. Rep., 1997.

[8] A. Kulkarni and S. Lodha, "Software protection through code obfuscation," *Project Report, Tata Research Development and Design Centre, Pune*, 2012.

[9] A. Kulkarni and R. Metta, "A new code obfuscation scheme for software protection," in *Proc. of the 8th IEEE International Symposium on Service Oriented System Engineering, SOSE 2014*. Washington, DC, USA: IEEE, 2014, pp. 409–414.

[10] H. Sajnani, V. Saini, and C. Lopes, "A parallel and efficient approach to large scale clone detection," *Journal of Software: Evolution and Process*, vol. 27, no. 6, pp. 402–429, 2015.

[11] H. Sajnani, R. Naik, and C. Lopes, "Application architecture discovery-towards domain-driven, easily-extensible code structure," in *Proc. of the 18th IEEE Working Conference on Reverse Engineering, WCRE-2011*. IEEE, 2011, pp. 401–405.

[12] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of computer programming*, vol. 74, no. 7, pp. 470–495, 2009.

[13] R. Lipton, "Fault diagnosis of computer programs. student report," 1971.

[14] J. Offutt, "A mutation carol: Past, present and future," *Information and Software Technology*, vol. 53, no. 10, pp. 1098–1107, 2011.

[15] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Proc. of the 2nd IEEE Working Conference on Reverse Engineering, WCRE 1995*. Toronto, Ontario, Canada: IEEE, 1995, pp. 86–95.

[16] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proceedings of the IEEE International Conference on Software Maintenance, ICSM'99*. Oxford, UK: IEEE, 1999, pp. 109–118.

[17] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics." in *Proc. of 1st IEEE International Conference on Software Maintenance, ICSM 1996*, vol. 96. Monterey, CA: IEEE, 1996, p. 244.

[18] K. A. Kontogiannis, R. DeMori, E. Merlo, M. Galler, and M. Bernstein, "Pattern matching for clone and concept detection," *Automated Software Engineering*, vol. 3, no. 1-2, pp. 77–108, 1996.

[19] B. Lague, D. Proulx, J. Mayrand, E. M. Merlo, and J. Hudepohl, "Assessing the benefits of incorporating function clone detection in a development process," in *Proc. of the IEEE International Conference on Software Maintenance, ICSM 1997*. IEEE, 1997, pp. 314–321.

[20] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," *Information and Software Technology*, vol. 55, no. 7, pp. 1165–1199, 2013.

[21] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queens School of Computing TR*, vol. 541, no. 115, pp. 64–68, 2007.

[22] A. Sheneamer and J. Kalita, "A survey of software clone detection techniques," *International Journal of Computer Applications*, vol. 137, no. 10, pp. 1–21, 2016.

[23] J. R. Gosler, "Software protection: Myth or reality?" in *Proc. of the Conference on the Theory and Application of Cryptographic Techniques*. Heidelberg, Berlin: Springer, 1985, pp. 140–157.

[24] A. Herzberg and S. S. Pinter, "Public protection of software," *ACM Transactions on Computer Systems (TOCS)*, vol. 5, no. 4, pp. 371–393, 1987.

[25] Y. Z. Rufai and A. Najib, "Computer security: A literature review and classification," *International Journal of Computer Science and Control Engineering*, vol. 4, no. 2, pp. 6–13, 2016.

[26] G. McGraw, "Technology transfer: A software security marketplace case study," *IEEE software*, vol. 28, no. 5, pp. 9–11, 2011.

[27] S. T. Kent, "Protecting externally supplied software in small computers." MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE, Tech. Rep., 1980.

[28] W. F. Zhu, "Concepts and techniques in software watermarking and obfuscation," Ph.D. dissertation, ResearchSpace@ Auckland, 2007.

[29] P. C. Van Oorschot, "Revisiting software protection," in *Proc. of the International Conference on Information Security, ICIS 2003*. Heidelberg, Berlin: Springer, 2003, pp. 1–13.

[30] G. Naumovich and N. Memon, "Preventing piracy, reverse engineering, and tampering," *Computer*, vol. 36, no. 7, pp. 64–71, 2003.

[31] C. Collberg and C. Thomborson, "On the limits of software watermarking," Department of Computer Science, The University of Auckland, New Zealand, Tech. Rep., 1998.

[32] C. Collberg and C. Thomborson, "Software watermarking: Models and dynamic embeddings," in *Proc. of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1999, pp. 311–324.

[33] T. Park and K. G. Shin, "Soft tamper-proofing via program integrity verification in wireless sensor networks," *IEEE Transactions on mobile computing*, vol. 4, no. 3, pp. 297–309, 2005.

[34] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (im) possibility of obfuscating programs," *Journal of the ACM (JACM)*, vol. 59, no. 2, p. 6, 2012.

[35] V. Balachandran and S. Emmanuel, "Potent and stealthy control flow obfuscation by stack based self-modifying code," *IEEE Transactions on Information Forensics and Security*, vol. 8, no. 4, pp. 669–681, 2013.

[36] G. Wroblewski, "General method of program code obfuscation (draft)," *Citeseer*, 2002.

[37] J. Cappaert, B. Wyseur, and B. Preneel, "Software security techniques [internal report, cosic]," 2004.

[38] C. J. Kapser and M. W. Godfrey, "Supporting the analysis of clones in software systems," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18, no. 2, pp. 61–82, 2006.

[39] C. J. Kapser and M. W. Godfrey, "cloning considered harmful considered harmful: patterns of cloning in software," *Empirical Software Engineering*, vol. 13, no. 6, p. 645, 2008.

[40] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics." in *Proc. of the 12th IEEE International Conference on Software Maintenance, ICSM 1996)*. Monterey, CA, USA: IEEE, 1996, pp. 244–253.

[41] A. Monden, D. Nakae, T. Kamiya, S.-i. Sato, and K.-i. Matsumoto, "Software quality analysis by code clones in industrial legacy software," in *Proc. of the 8th IEEE Symposium on Software Metrics*. Ottawa, Canada: IEEE, 2002, pp. 87–94.

[42] J. H. Johnson, "Substring matching for clone detection and change tracking." in *Proc. of the 10th international Conference on Software Maintenance, ICSM 1994*, vol. 94, Victoria, British Columbia, Canada, 1994, pp. 120–126.

[43] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Evaluating the harmfulness of cloning: A change based experiment," in *Proc. of the 4th International Workshop on Mining Software Repositories*. Minneapolis, MN, USA: IEEE Computer Society, 2007, p. 18.

[44] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[45] T. Lavoie, M. Eilers-Smith, and E. Merlo, "Challenging cloning related problems with gpu-based algorithms," in *Proc. of the 4th International Workshop on Software Clones*. Cape Town, SA: ACM, 2010, pp. 25–32.

[46] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An ethnographic study of copy and paste programming practices in oopl," in *Proc. of the 3rd International Symposium*

*on Empirical in Software Engineering, ISESE 2004*. Redondo Beach, CA, USA: ACM-IEEE, 2004, pp. 83–92.

[47] L. Jiang, Z. Su, and E. Chiu, "Context-based detection of clone-related bugs," in *Proc. of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. Dubrovnik, Croatia: ACM, 2007, pp. 55–64.

[48] 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*, vol. 32, no. 3, pp. 176–192, 2006.

[49] J. Krinke, N. Gold, Y. Jia, and D. Binkley, "Cloning and copying between gnome projects," in *Proc. of the 7th IEEE Working Conference on Mining Software Repositories, MSR 2010*. Cape Town, SA: IEEE, 2010, pp. 98–101.

[50] B. S. Baker, "A program for identifying duplicated code," *Computing Science and Statistics*, pp. 49–49, 1993.

[51] B. S. Baker, "Parameterized difference," in *Proc. of the 10th ACM-SIAM Symposium on Discrete Algorithms, SODA 1999*. Maryland, USA: ACM, 1999, pp. 854–855.

[52] R. Koschke, R. Falke, and P. Frenzel, "Clone detection using abstract syntax suffix trees," in *Proc. of the 13th Working Conference on Reverse Engineering, WCRE 2006*. Italy: IEEE, 2006, pp. 253–262.

[53] J. H. Johnson, "Identifying redundancy in source code using fingerprints," in *Proc. of the conference of the Centre for Advanced Studies on Collaborative research: software engineering-Volume 1*. Canada: IBM Press, 1993, pp. 171–183.

[54] I. J. Cox and J.-P. Linnartz, "Some general methods for tampering with watermarks," *IEEE Journal on selected areas in communications*, vol. 16, no. 4, pp. 587–593, 1998.

[55] J. R. Cordy, T. R. Dean, and N. Synytskyy, "Practical language-independent detection of near-miss clones," in *Proc. of the conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 2004, pp. 1–12.

[56] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proc. of the IEEE International Conference on Software Maintenance, ICSM 1999*. IEEE, 1999, pp. 109–118.

[57] S. T. Dumais *et al.*, "Latent semantic indexing (lsi) and trec-2," *Nist Special Publication Sp*, pp. 105–105, 1994.

[58] A. Marcus and J. I. Maletic, "Identification of high-level concept clones in source code," in *Proc. of the 16th Annual International Conference on Automated Software Engineering, ASE 2001*. Washington, DC, USA: IEEE, 2001, pp. 107–114.

[59] 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*, vol. 28, no. 7, pp. 654–670, 2002.

[60] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: A tool for finding copy-paste and related bugs in operating system code." in *OSdi*, vol. 4, no. 19, 2004, pp. 289–302.

[61] 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*, vol. 32, no. 3, pp. 176–192, 2006.

[62] E. Juergens, F. Deissenboeck, and B. Hummel, "Clonedetective-a workbench for clone detection research," in *Proc. of the 31st International Conference on Software Engineering*. Vancouver, BC, Canada: IEEE Computer Society, 2009, pp. 603–606.

[63] S. Kawaguchi, T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, and H. Iida, "Shinobi: A tool for automatic code clone detection in the ide," in *Proc. of the 16th IEEE Working Conference on Reverse Engineering, WCRE-2009*, Lille, France, 2009, pp. 313–314.

[64] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proc. of the 14th International Conference on Software Maintenance, ICSM1998.* Maryland, USA: IEEE, 1998, pp. 368–377.

[65] A. Raza, G. Vogel, and E. Plödereder, "Bauhaus–a tool suite for program analysis and reverse engineering," in *International Conference on Reliable Software Technologies.* Heidelberg, Berlin: Springer, 2006, pp. 71–82.

[66] W. Yang, "Identifying syntactic differences between two programs," *Software: Practice and Experience*, vol. 21, no. 7, pp. 739–755, 1991.

[67] V. Wahler, D. Seipel, J. Wolff, and G. Fischer, "Clone detection in source code by frequent itemset techniques," in *Proc. of the 4th IEEE International Workshop on Source Code Analysis and Manipulation, WSCAM 2004.* USA: IEEE, 2004, pp. 128–135.

[68] B. S. Baker, "Finding clones with dup: Analysis of an experiment," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 608–621, 2007.

[69] W. S. Evans, C. W. Fraser, and F. Ma, "Clone detection via structural abstraction," *Software Quality Journal*, vol. 17, no. 4, pp. 309–330, 2009.

[70] E. Duala-Ekoko and M. P. Robillard, "Clonetracker: tool support for code clone management," in *Proc. of the 30th international conference on Software engineering.* Washington, DC, USA: ACM, 2008, pp. 843–846.

[71] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, "Clone management for evolving software," *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 1008–1026, 2012.

[72] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *International static analysis symposium.* Heidelberg, Berlin: Springer, 2001, pp. 40–56.

[73] J. Krinke, "Identifying similar code with program dependence graphs," in *Proc. of the 8th Working Conference on Reverse Engineering, WCRE 2001.* Germany: IEEE, 2001, pp. 301–309.

[74] C. Liu, C. Chen, J. Han, and P. S. Yu, "Gplag: detection of software plagiarism by program dependence graph analysis," in *Proc. of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining, KDD 2006.* Philadelphia, USA: ACM, 2006, pp. 872–881.

[75] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[76] R. V. Komondoor and S. Horwitz, "Automated duplicated-code detection and procedure extraction," Ph.D. dissertation, Citeseer, 2003.

[77] K. Gallagher and L. Layman, "Are decomposition slices clones?" in *11th IEEE International Workshop on Program Comprehension.* IEEE, 2003, pp. 251–256.

[78] G. A. Di Lucca, M. Di Penta, A. R. Fasolino, and P. Granato, "Clone analysis in the web era: An approach to identify cloned web pages," in *Proc. of the 7th IEEE Workshop on Empirical Studies of Software Maintenance*, Italy, 2001, pp. 107–113.

[79] G. A. Di Lucca, M. Di Penta, and A. R. Fasolino, "An approach to identify duplicated web pages," in *Proc. of the 26th Annual International Conference on Computer Software and Applications, COMPSAC 2002.* England: IEEE, 2002, pp. 481–486.

[80] F. Calefato, F. Lanubile, and T. Mallardo, "Function clone detection in web applications: a semiautomated approach," *J. Web Eng.*, vol. 3, no. 1, pp. 3–21, 2004.

[81] F. Lanubile and T. Mallardo, "Finding function clones in web applications," in *Proc. of the 7th European Conference on Software Maintenance and Reengineering.* IEEE, 2003, pp. 379–386.

[82] R. Tairas and J. Gray, "Phoenix-based clone detection using suffix trees," in *Proc. of the 44th annual Southeast regional conference, ACM-SE 2006.* Melbourne: ACM, 2006, pp. 679–684.

[83] K. Greenan, "Method-level code clone detection on transformed abstract syntax trees using sequence matching algorithms," *Student Report, University of California-Santa Cruz, Winter*, 2005.

[84] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proc. of the 29th international conference on Software Engineering, ICSE 2007.* IEEE Computer Society, 2007, pp. 96–105.

[85] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proc. of the 20th annual symposium on Computational geometry, SoCG 2004.* New York, NY, USA: ACM, 2004, pp. 253–262.

[86] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis, "Measuring clone based reengineering opportunities," in *Proc. of the 6th International software metrices symposium, METRICS 1999.* Florida, USA: IEEE, 1999, pp. 292–303.

[87] J.-F. Patenaude, E. Merlo, M. Dagenais, and B. Laguë, "Extending software quality assessment techniques to java systems," in *Proc. of the 7th International Workshop on Program Comprehension, IWPC 1999.* USA: IEEE, 1999, pp. 49–56.

[88] M. De Wit, A. Zaidman, and A. Van Deursen, "Managing code clones using dynamic change tracking and resolution," in *Proc. of the IEEE International Conference on Software Maintenance, ICSM 2009.* Edmonton, AB: IEEE, 2009, pp. 169–178.

[89] R. V. Patil, S. D. Joshi, S. V. Shinde, D. A. Ajagekar, and S. D. Bankar, "Code clone detection using decentralized architecture and code reduction," in *Proc. of the IEEE International Conference on Pervasive Computing, ICPC-2015.* Pune, India: IEEE, 2015, pp. 1–6.

[90] I. Keivanloo, F. Zhang, and Y. Zou, "Threshold-free code clone detection for a large-scale heterogeneous java repository," in *Proc. of the 22nd IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2015*. IEEE, 2015, pp. 201–210.

[91] S. Chodarev, E. Pietriková, and J. Kollár, "Haskell clone detection using pattern comparing algorithm," in *13th International Conference on Engineering of Modern Electric Systems, EMES 2015*. IEEE, 2015, pp. 1–4.

[92] T. Kamiya, "An execution-semantic and content-and-context-based code-clone detection and analysis," in *9th International Workshop on Software Clones, IWSC 2015*. IEEE, 2015, pp. 1–7.

[93] M. Singh and V. Sharma, "Detection of file level clone for high level cloning," *Procedia Computer Science*, vol. 57, pp. 915–922, 2015.

[94] H. A. Basit and S. Jarzabek, "A data mining approach for detecting higher-level clones in software," *IEEE Transactions on Software engineering*, no. 4, pp. 497–514, 2009.

[95] S. Goldwasser and G. N. Rothblum, "On best-possible obfuscation," *Journal of Cryptology*, vol. 27, no. 3, pp. 480–505, 2014.

[96] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl, "Protecting software through obfuscation: Can it keep pace with progress in code analysis?" *ACM Computing Surveys (CSUR)*, vol. 49, no. 1, p. 4, 2016.

[97] A. Balakrishnan and C. Schulze, "Code obfuscation literature survey: Cs701construction of compilers," *Instructor: Charles Fischer Computer Sciences Department University of Wisconsin, Madison December*, vol. 19, 2005.

[98] N. P. Varnovsky and V. A. Zakharov, "On the possibility of provably secure obfuscating programs," in *Proc. of the International Andrei Ershov Memorial Conference on Perspectives of System Informatics*. Russia: Springer, 2003, pp. 91–102.

[99] B. Lynn, M. Prabhakaran, and A. Sahai, "Positive results and techniques for obfuscation," in *Proc. of the International conference on the theory and applications of cryptographic techniques*. Switzerland: Springer, 2004, pp. 20–39.

[100] M. Ceccato, M. Di Penta, J. Nagra, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella, "The effectiveness of source code obfuscation: An experimental assessment," in *Proc. of the 17th IEEE International Conference on Program Comprehension, ICPC 2009*. IEEE, 2009, pp. 178–187.

[101] D. Low, "Java control flow obfuscation," Ph.D. dissertation, Citeseer, 1998.

[102] C. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *Proc. of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages,*. New York, NY, USA: ACM, 1998, pp. 184–196.

[103] C. Wang, J. Davidson, J. Hill, and J. Knight, "Protection of software-based survivability mechanisms," in *Proc. of the International Conference on Dependable Systems and Networks, DSN 2001*. Washington, DC, USA: IEEE, 2001, pp. 193–202.

[104] M. Madou, B. Anckaert, B. De Sutter, and K. De Bosschere, "Hybrid static-dynamic attacks against software protection mechanisms," in *Proc. of the 5th ACM workshop on Digital rights management, WDRM 2005*. New York, NY, USA: ACM, 2005, pp. 75–82.

[105] F. Cohen, "Computer viruses," *Computers & security*, vol. 6, no. 1, pp. 22–35, 1987.

[106] F. B. Cohen, "Operating system protection through program evolution." *Computers & Security*, vol. 12, no. 6, pp. 565–584, 1993.

[107] "Dans tools," 2009. [Online]. Available: http://www.danstools.com/javascript-obfuscate/index.php

[108] "Daft logic," 2009. [Online]. Available: https://www.daftlogic.com/projects-online-javascript-obfuscator.html

[109] "Packers," 2008. [Online]. Available: http://packer.50x.eu/

[110] "Eclipse," 2010. [Online]. Available: http://www.eclipse.org/downloads/packages/eclipse-ide-java-and-dsl-developers/mars2

[111] P. Gautam and H. Saini, "Various code clone detection techniques and tools: a comprehensive survey," in *International Conference on Smart Trends for Information Technology and Computer Communications*. Singapore: Springer, 2016, pp. 655–667.

[112] D. Chatterji, J. C. Carver, N. A. Kraft, and J. Harder, "Effects of cloned code on software maintainability: A replicated developer study," in *20th Working Conference on Reverse Engineering, WCRE 2013*. IEEE, 2013, pp. 112–121.

[113] D. Chatterji, J. C. Carver, B. Massengil, J. Oslin, and N. A. Kraft, "Measuring the efficacy of code clone information in a bug localization task: An empirical study," in *International Symposium on Empirical Software Engineering and Measurement, ESEM 2011*. IEEE, 2011, pp. 20–29.

[114] C. K. Roy and J. R. Cordy, "An empirical study of function clones in open source software," in *15th Working Conference on Reverse Engineering, 2008*. IEEE, 2008, pp. 81–90.

[115] C. K. Roy and J. R. Cordy, "Near-miss function clones in open source software: an empirical study," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 22, no. 3, pp. 165–189, 2010.

[116] P. Gautam and H. Saini, "Non-trivial software clone detection using program dependency graph," *International Journal of Open Source Software and Processes (IJOSSP)*, vol. 8, no. 2, pp. 1–24, 2017.

[117] R. Ami and H. Haga, "Code clone detection method based on the combination of tree-based and token-based methods," *Journal of Software Engineering and Applications*, vol. 10, no. 13, p. 891, 2017.

[118] M. A. Nishi and K. Damevski, "Scalable code clone detection and search based on adaptive prefix filtering," *Journal of Systems and Software*, vol. 137, pp. 130–142, 2018.

[119] E. Kodhai, S. Kanmani, A. Kamatchi, R. Radhika, and B. V. Saranya, "Detection of type-1 and type-2 code clones using textual analysis and metrics," in *International Conference on Recent Trends in Information, Telecommunication and Computing, ITC 2010*. IEEE, 2010, pp. 241–243.

[120] P. Gautam and H. Saini, "A hybrid approach for detection of type-1 software clones," in *4th International Conference on Signal Processing, Computing and Control, ISPCC 2017*. IEEE, 2017, pp. 279–282.

[121] S. Shafieian and Y. Zou, "Comparison of clone detection techniques," Technical report, Queen, Tech. Rep., 2012.

[122] "Wet lab," 1989. [Online]. Available: http://ftp.gnu.org/gnu/wget/

[123] "Codeblock," 2016. [Online]. Available: www.codeblocks.org/

[124] "Pmd," 2007. [Online]. Available: http://www.pmd.sourcegefourge.net.

[125] T. Kamiya, "The official ccfinderx website," *URL http://www. ccfinder. net/ccfinderx. html Last accessed November*, 2008.

[126] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: A tool for finding copy-paste and related bugs in operating system code." in *Proc. of the 6th conference on Symposium on Operating Systems Design & Implementation, OSDI 2004*, vol. 4, no. 19. Berkeley, CA, USA: ACM, 2004, pp. 289–302.

[127] S. Bellon, "Vergleich von techniken zur erkennung duplizierten quellcodes," *Master's Thesis, Institut fur Softwaretechnologie, Universitat Stuttgart, Stuttgart, Germany*, 2002.

[128] P. Gautam and H. Saini, "Type-2 software cone detection using directed acyclic graph," in *4th International Conference on Image Information Processing, ICIIP 2017*. IEEE, 2017, pp. 1–4.

[129] E. Malmi, N. Tatti, and A. Gionis, "Beyond rankings: comparing directed acyclic graphs," *Data mining and knowledge discovery*, vol. 29, no. 5, pp. 1233–1257, 2015.

[130] "Dagitty." [Online]. Available: http://www.dagitty.net/dags.html

[131] R. Tekchandani, R. Bhatia, and M. Singh, "Semantic code clone detection for internet of things applications using reaching definition and liveness analysis," *The Journal of Supercomputing*, pp. 1–28, 2016.

[132] Y. Sharma, "Hybrid technique for object oriented software clone detection," Ph.D. dissertation, 2011.

[133] A. Puri and S. Kumar, "Software code clone detection model," *International Journal of Computers and Distributed Systems*, vol. 1, no. 3, pp. 69–74, 2012.

[134] R. Tekchandani, R. K. Bhatia, and M. Singh, "Semantic code clone detection using parse trees and grammar recovery," 2013.

[135] J. Svajlenko and C. K. Roy, "Cloneworks: A fast and flexible large-scale near-miss clone detection tool," in *Proc. of the 39th International Conference on Software Engineering Companion*. IEEE Press, 2017, pp. 177–179.

[136] C. Ragkhitwetsagul and J. Krinke, "Using compilation/decompilation to enhance clone detection," in *Proc. of the 11th International Workshop on Software Clone, IWSC 2017*, vol. 11. IEEE, 2017, pp. 8–14.

[137] S. Gupta *et al.*, "Detection of near-miss clones using metrics and abstract syntax trees," in *Proc. of the International Conference on Inventive Communication and Computational Technologies, ICICCT 2017*. IEEE, 2017, pp. 230–234.

[138] Y. Sabi, Y. Higo, and S. Kusumoto, "Rearranging the order of program statements for code clone detection," in *Proc. of the 11th International Workshop on Software Clones, IWSC 2017*. IEEE, 2017, pp. 1–7.

[139] M. Sudhamani and L. Rangarajan, "Code clone detection based on order and content of control statements," in *Proc. of the 2nd International Conference on Contemporary Computing and Informatics, IC3I 2016*. IEEE, 2016, pp. 59–64.

[140] J. Laski and W. Szermer, "Identification of program modifications and its applications in software maintenance," in *Proc. of the Conference on Software Maintenance*. IEEE, 1992, pp. 282–290.

[141] B. A. Cota, D. G. Fritz, and R. G. Sargent, "Control flow graphs as a representation language," in *Proc. of the 26th conference on Winter simulation*. Society for Computer Simulation International, 1994, pp. 555–559.

[142] M. J. Harrold, G. Rothermel, and A. Orso, "Representation and analysis of software," *Lecture Notes*, 2005.

[143] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[144] A. H. Watson, D. R. Wallace, and T. J. McCabe, *Structured testing: A testing methodology using the cyclomatic complexity metric*. US Department of Commerce, Technology Administration, National Institute of Standards and Technology, 1996, vol. 500, no. 235.

[145] "Wet lab," 1989. [Online]. Available: http://ftp.gnu.org/gnu/wget/

[146] R. Tairas and J. Gray, "Phoenix-based clone detection using suffix trees," in *Proc. of the 44th Annual Southeast regional conference*. New York, NY, USA: ACM, 2006, pp. 679–684.

[147] C. K. Roy, "Detection and analysis of near-miss software clones," in *Proc. of the IEEE International Conference on Software Maintenance. ICSM 2009*. IEEE, 2009, pp. 447–450.

[148] R. Koschke, "Survey of research on software clones," in *Proc. of the Dagstuhl Seminar*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007, pp. 1–24.

[149] R. Koschke, "Frontiers of software clone management," in *Proc. of the IEEE International Conference of Frontiers of Software Maintenance, FoSM 2008.* IEEE, 2008, pp. 119–128.

[150] A. Kaur and M. S. Sandhu, "Software code clone detection model using hybrid approach," *INTERNATIONAL JOURNAL OF COMPUTERS & TECHNOLOGY*, vol. 3, no. 2b, pp. 275–278, 2012.

[151] B. R. Kaur M., Rattan D. and S. M., "Comparison and evaluation of clone detection tools: An experimental approach," *CSI Journal of computing*, vol. 1, no. 4, 2012.

[152] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on software engineering*, vol. 33, no. 9, 2007.

[153] F. Arcelli Fontana, M. Zanoni, A. Ranchetti, and D. Ranchetti, "Software clone detection and refactoring," *ISRN Software Engineering*, vol. 2013, 2013.

[154] S. Dang and S. A. Wani, "Performance evaluation of clone detection tools," *INTERNATIONAL JOURNAL OF SCIENCE AND RESEARCH (IJSR)*, pp. 1903–1906, 2015.

[155] S. A. Wani and S. Dang, "A comparative study of clone detection tools," 2015.

[156] K. Kaur and R. Maini, "A comprehensive review of code clone detection techniques," *vol. IV, no*, vol. 12, pp. 43–47, 2015.

[157] K. Solanki and S. Kumari, "Comparative study of software clone detection techniques," in *International Conference on Management and Innovation Technology, MITicon 2016*. IEEE, 2016, pp. MIT–152.

[158] H. Kaur and R. Maini, "Performance evaluation and comparative analysis of code-clone-detection techniques and tools," 2017.

[159] C. K. Roy and J. R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *Proc. of the 16th IEEE International Conference on Program Comprehension, ICPC 2008*. Washington, DC, USA: IEEE, 2008, pp. 172–181.

[160] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, "Index-based code clone detection: incremental, distributed, scalable," in *IEEE International Conference on Software Maintenance, ICSM 2010*. Timioara, Romania: IEEE, 2010, pp. 1–9.

[161] S. Lee and I. Jeong, "Sdd: high performance code clone detection system for large scale source code," in *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM, 2005, pp. 140–141.

[162] P. Bulychev and M. Minea, "Duplicate code detection using anti-unification," in *Proc. of the Spring/Summer Young Researchers Colloquium on Software Engineering*, no. 2. , 2008.

[163] M. Bruntink, A. Van Deursen, R. Van Engelen, and T. Tourwe, "On the use of clone detection for identifying crosscutting concern code," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 804–818, 2005.

[164] E. Burd and J. Bailey, "Evaluating clone detection tools for use during preventative maintenance," in *Proceedings of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation*. Washington, DC, USA: IEEE, 2002, pp. 36–43.

[165] F. V. Rysselberghe and S. Demeyer, "Evaluating clone detection techniques from a refactoring perspective," in *Proc. of the 19th IEEE international conference on Automated software engineering*. IEEE Computer Society, 2004, pp. 336–339.

[166] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proc. of the 27th international conference on Software engineering*. St. Louis, MO, USA: ACM, 2005, pp. 402–411.

[167] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[168] R. G. Hamlet, "Testing programs with the aid of a compiler," *IEEE Transactions on Software engineering*, no. 4, pp. 279–290, 1977.

[169] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 5, no. 2, pp. 99–118, 1996.

[170] R. A. DeMillo, "Completely validated software: test adequacy and program mutation (panel session)," in *Proc. of the 11th international conference on Software engineering*. ACM, 1989, pp. 355–356.

[171] A. J. Offutt, "Investigations of the software testing coupling effect," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 1, no. 1, pp. 5–20, 1992.

[172] P. G. Frankl and S. N. Weiss, "An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria," in *Proc. of the 4th symposium on Testing, analysis, and verification*. New York, NY, USA: ACM, 1991, pp. 154–164.

[173] P. Thevenod-Fosse, H. Waeselynck, and Y. Crouzet, "An experimental study on software structural testing: deterministic versus random input generation," in *Proc. of the 21st International Symposium on Fault-Tolerant compouting*. Montreal, Canada: IEEE, 1991, pp. 410–417.

[174] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria," in *Proc. of the 16th international conference on Software engineering*. Sorrento Italy: IEEE Computer Society Press, 1994, pp. 191–200.

[175] S.-W. Kim, J. A. Clark, and J. A. McDermid, "Investigating the effectiveness of object-oriented testing strategies using the mutation method," *Software Testing, Verification and Reliability*, vol. 11, no. 4, pp. 207–225, 2001.

[176] A. Memon, I. Banerjee, and A. Nagarajan, "What test oracle should i use for effective gui testing?" in *Proc. of the 18th IEEE International Conference on Automated Software Engineering, ASE-2003*. Montreal, Quebec, Canada: IEEE, 2003, pp. 164–173.

[177] J. H. Andrews and Y. Zhang, "General test result checking with log file analysis," *IEEE Transactions on Software Engineering*, vol. 29, no. 7, pp. 634–648, 2003.

[178] L. C. Briand, Y. Labiche, and Y. Wang, "Using simulation to empirically investigate test coverage criteria based on statechart," in *Proc. of the 26th International Conference on Software Engineering, ICSE 2004*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 86–95.

[179] W. Chen, R. H. Untch, G. Rothermel, S. Elbaum, and J. Von Ronne, "Can fault-exposure-potential estimates improve the fault detection abilities of test suites?" *Software Testing, Verification and Reliability*, vol. 12, no. 4, pp. 197–218, 2002.

[180] A. P. Mathur, "Performance, effectiveness, and reliability issues in software testing," in *15th Annual International Computer Software & Applications Conference, ICSAC 1991*. IEEE, 1991, pp. 604–605.

[181] H. Agrawal, R. DeMillo, R. Hathaway, W. Hsu, W. Hsu, E. Krauser, R. J. Martin, A. Mathur, and E. Spafford, "Design of mutant operators for the c programming language," Technical Report SERC-TR-41-P, Software Engineering Research Center, Department of Computer Science, Purdue University, Indiana, Tech. Rep., 1989.

[182] M. E. Delamaro, J. C. Maldonado, and A. Mathur, "Proteum-a tool for the assessment of test adequacy for c programs users guide," in *PCS*, vol. 96, 1996, pp. 79–95.

[183] C. K. Roy and J. R. Cordy, "Scenario-based comparison of clone detection techniques," in *The 16th IEEE International Conference on Program Comprehension*. IEEE, 2008, pp. 153–162.

[184] C. K. Roy and J. R. Cordy, "A mutation/injection-based automatic framework for evaluating code clone detection tools," in *IEEE International Conference on Software Testing, Verification, and Validation Workshops*. Washington, DC: IEEE, 2009, pp. 157–166.

[185] C. K. Roy and J. R. Cordy, "Towards a mutation-based automatic framework for evaluating code clone detection tools," in *Proc. of the 2008 C 3 S 2 E conference*. New York, NY, USA: ACM, 2008, pp. 137–140.

[186] T. Budd and F. Sayward, "Users guide to the pilot mutation system," *Yale University, New Haven, Connecticut, Technique Report*, vol. 114, 1977.

[187] M. R. Woodward, "Mutation testing-an evolving technique," in *IEE Colloquium on Software Testing for Critical Systems,*. IET, 1990, pp. 3–1.

[188] M. R. Woodward, "Mutation testingits origin and evolution," *Information and Software Technology*, vol. 35, no. 3, pp. 163–169, 1993.

[189] A. P. Mathur, "Foundations of software testing," 2008.

[190] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2016.

[191] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE transactions on software engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[192] Q. Zhu, A. Panichella, and A. Zaidman, "A systematic literature review of how mutation testing supports test activities," PeerJ Preprints, Tech. Rep., 2016.

[193] P. Reales, M. Polo, J. L. Fernandez-Aleman, A. Toval, and M. Piattini, "Mutation testing," *IEEE software*, vol. 31, no. 3, pp. 30–35, 2014.

[194] M. Gligoric, V. Jagannath, Q. Luo, and D. Marinov, "Efficient mutation testing of multithreaded code," *Software Testing, Verification and Reliability*, vol. 23, no. 5, pp. 375–403, 2013.

[195] R. Baker and I. Habli, "An empirical evaluation of mutation testing for improving the test quality of safety-critical software," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 787–805, 2013.

[196] P. R. Mateo and M. P. Usaola, "Reducing mutation costs through uncovered mutants," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 464–489, 2015.

[197] A. S. Namin, X. Xue, O. Rosas, and P. Sharma, "Muranker: a mutant ranking tool," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 572–604, 2015.

[198] A. Derezinska and K. Hałas, "Improving mutation testing process of python programs," in *Software Engineering in Intelligent Systems*. Springer, 2015, pp. 233–242.

[199] C.-a. Sun, F. Xue, H. Liu, and X. Zhang, "A path-aware approach to mutant reduction in mutation testing," *Information and Software Technology*, vol. 81, pp. 65–81, 2017.

[200] P. Delgado-Pérez, I. Medina-Bulo, F. Palomo-Lozano, A. García-Domínguez, and J. J. Domínguez-Jiménez, "Assessment of class mutation operators for c++ with the mucpp mutation system," *Information and Software Technology*, vol. 81, pp. 169–184, 2017.

[201] M. B. Bashir and A. Nadeem, "Improved genetic algorithm to reduce mutation testing cost," *IEEE Access*, vol. 5, pp. 3657–3674, 2017.

[202] P. Gautam and H. Saini, "A mutation operator-based scenario for evaluating software-clone detection tools and techniques", International Journal of Information Security and Privacy (IJISP), vo.l13,no.1, 2017. (in press)

[203] P. Gautam and H. Saini, "Correlation and performance estimation of clone detection tools", International Journal of Open Source Software and Processes (IJOSSP), vol.9, no. 2, pp. 55-71, 2018.

[204] P. Gautam and H. Saini, "A novel software protection approach for code obfuscation to enhance software security", International Journal of Mobile Computing and Multimedia Communications (IJMCMC),vol.8, no. 1, pp. 34-47, 2017.

# LIST OF PUBLICATIONS

**Published Journal Papers**

1. P. Gautam and H. Saini, "A Novel Software Protection Approach for Code Obfuscation to Enhance Software Security", *International Journal of Mobile Computing and Multimedia Communications (IJMCMC),*vol.8, no. 1, pp. 34-47, 2017.

   *[Major indexing: SCOPUS, DBLP, ACM digital Library, web of sciences, Google scholar].*

2. P. Gautam and H. Saini. "Non-Trivial Software Clone Detection Using Program Dependency Graph", *International Journal of Open Source Software and Procsses (IJOSSP),* vol. 8, no.2, pp. 1-24, 2017.

   *[Major indexing: SCOPUS, DBLP, ACM digital Library, web of sciences, Google scholar].*

3. P. Gautam and H. Saini, "Correlation and Performance Estimation of Clone Detection Tools", *International Journal of Open Source Software and Processes (IJOSSP),* vol. 9, no. 2, pp. 55-71, 2018. .
   *[Major indexing: SCOPUS, DBLP, ACM digital Library, web of sciences, Google scholar]*

4. P. Gautam and H. Saini, " A Mutation Operator-Based Scenario for Evaluating Software Clone Detection Tools and Techniques, *International Journal of Information Security and Privacy (IJISP),* vol. 13, no.1, pp. 30-45, 2019.

   *[Major indexing: SCOPUS, ESCI, DBLP, ACM digital Library, web of sciences, Google scholar].*

**Accepted Journal Papers**

5. P. Gautam and H. Saini, "Mutation Testing-based Evaluation Framework for Evaluating Software Clone Detection Tools", *International Journal of Life Cycle Reliability and Safety Engineering*. Springer. vol. xx, no. xx, pp. xx-xx, 20xx.

[Major Indexing: *Google Scholar, CNKI, EBSCO Discovery Service, OCLC, Summon by ProQuest]*

**Published Book Chapters**

6. P. Gautam and H. Saini(2016), "Various Code Clone Detection Techniques and Tools: A Comprehensive Survey", *Proceedings of the International conference on Smart Trends for Information and Knowledge computing,* (CCIS'16), Springer, Singapore, pp. 255-267

   *[Major indexing: SCOPUS, DBLP, Google scholar].*

**Published Conference Proceedings**

7. P. Gautam and H. Saini (2017), "A Hybrid Approach for Detection of Type-1 Software Clones*", In 4th International conference on Signal Processing, Computing and Control (ISPCC-*

*2017), IEEE,* Jaypee University of Information Technology, Waknaghat, Solan, HP, India, pp. 279-282.

*[Major indexing: SCOPUS, Conference Proceedings Citation Index]*

8. Pratiksha Gautam and Hemraj Saini(2018). "Type-2 Software Cone Detection Using Directed Acyclic Graph". *4th IEEE International conference on Image Information Processing (ICIIP - 2017),* Jaypee University of Information Technology, Waknaghat, Solan, HP, India, (December 2017), pp. 205-208.

*[Major indexing: SCOPUS, Conference Proceedings Citation Index]*

**Communicated Journal Papers**

9. P. Gautam and H. Saini, "Mutation-Based Editing Taxonomy of Different Types of Software Clones", *International Journal of Information Technology and Web Engineering* [Under Review].

*[Major indexing: SCOPUS, ESCI, DBLP, ACM digital Library, web of sciences, Google scholar].*

10. P. Gautam and H. Saini, "Detection of Renamed Clones (type-2) Using Directed Acyclic Graph" *International Journal of Information Technology.* Springer, Communicated [Under Review].
[Major Indexing*: INSPEC, Google Scholar, CNKI, DBLP, EBSCO Discovery Service, OCLC, Proquest]*

11. P. Gautam and H. Saini (2018**). "**Detection of Type-1 Software Clones Using a Hybrid Approach" *In International Journal of Information Technology.* Springer, Communicated [Under Review].

[Major Indexing*: INSPEC, Google Scholar, CNKI, DBLP, EBSCO Discovery Service, OCLC, Proquest]*

12. P. Gautam and H. Saini. "Software Security Concerns with Open Research Issues — A Review", *In ACM Journal of computing survey* [Communicated].

*[Major indexing: SCIE, SCOPUS, ESCI, DBLP, ACM digital Library, web of sciences, Google scholar].*

*Candidate's Signature*