Lingxiao Jiang
December 2009
Computer Science

Scalable Detection of Similar Code: Techniques and Applications

Abstract

Similar code, also known as cloned code, commonly exists in large software. Studies show that code duplication can incur higher software maintenance cost and more software defects. Thus, detecting similar code and tracking its migration have many important applications, including program understanding, refactoring, optimization, and bug detection.

This dissertation presents novel, general techniques for detecting and analyzing both syntactic and semantic code clones. The techniques can scalably and accurately detect clones based on various similarity definitions, including trees, graphs, and functional behavior. They also have the general capability to help reduce software defects and advance code reuse. Specifically, this dissertation makes the following main contributions:

First, it presents DECKARD, a tree-based clone detection technique and tool. The key insight is that we accurately represent syntax trees and dependency graphs of a program as characteristic vectors in the Euclidean space and apply hashing algorithms to cluster similar vectors efficiently. Experiments show that DECKARD scales to millions of lines of code with few false positives. In addition, DECKARD is language-agnostic and easily parallelizable, with the potential to scale to billions of lines of code in different languages.

Second, it describes a novel application of DECKARD to bug detection. In particular, it introduces a general notion of *context-based inconsistencies* as indicators of clone-related bugs and formalizes three concrete types of such inconsistencies. The formalization is then applied to the clones identified by DECKARD, and many previously unknown bugs in large projects are discovered. These bugs exhibit diverse characteristics and cannot be detected by any single previous bug detection technique.

Third, the disseration presents EQMINER, a practical technique to detect functionally

equivalent code. Inspired by Schwartz's randomized polynomial identity testing, EqMiner adapts automated random testing in a novel way to quickly determine functional equivalence among arbitrary code fragments automatically extracted from a large program. Evaluated on the Linux kernel, EqMiner discovered many functionally equivalent, but syntactically different code fragments, which can facilitate future studies on semantic-aware code reuse.

Scalable Detection of Similar Code: Techniques and Applications

By

Lingxiao Jiang
B.S. (Peking University) 2000
M.S. (Peking University) 2003

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Professor Zhendong Su, Chair

Professor Premkumar T. Devanbu

Professor Ronald A. Olsson

Doctor Daniel J. Quinlan

Committee in Charge
2009

Scalable Detection of Similar Code: Techniques and Applications

To my dear wife Xiaojing and our parents.

Contents

List of Figures

List of Tables

Abstract

Similar code, also known as cloned code, commonly exists in large software. Studies show that code duplication can incur higher software maintenance cost and more software defects. Thus, detecting similar code and tracking its migration have many important applications, including program understanding, refactoring, optimization, and bug detection.

This dissertation presents novel, general techniques for detecting and analyzing both syntactic and semantic code clones. The techniques can scalably and accurately detect clones based on various similarity definitions, including trees, graphs, and functional behavior. They also have the general capability to help reduce software defects and advance code reuse. Specifically, this dissertation makes the following main contributions:

First, it presents DECKARD, a tree-based clone detection technique and tool. The key insight is that we accurately represent syntax trees and dependency graphs of a program as characteristic vectors in the Euclidean space and apply hashing algorithms to cluster similar vectors efficiently. Experiments show that DECKARD scales to millions of lines of code with few false positives. In addition, DECKARD is language-agnostic and easily parallelizable, with the potential to scale to billions of lines of code in different languages.

Second, it describes a novel application of DECKARD to bug detection. In particular, it introduces a general notion of *context-based inconsistencies* as indicators of clone-related bugs and formalizes three concrete types of such inconsistencies. The formalization is then applied to the clones identified by DECKARD, and many previously unknown bugs in large projects are discovered. These bugs exhibit diverse characteristics and cannot be detected by any single previous bug detection technique.

Third, the disseration presents EQMINER, a practical technique to detect functionally equivalent code. Inspired by Schwartz's randomized polynomial identity testing, EQMINER adapts automated random testing in a novel way to quickly determine functional equivalence among arbitrary code fragments automatically extracted from a large program. Evaluated on the Linux kernel, EQMINER discovered many functionally equivalent, but syntactically different code fragments, which can facilitate future studies on semantic-aware code reuse.

Acknowledgments and Thanks

Throughout my whole Ph.D. program, many people have played important roles in my memorable years at UC Davis. I would like to express my sincere gratitude and appreciation to them.

My major advisor, Professor Zhendong Su, is the most important person who helped to shape my research ideas and styles. He is very patient, tolerant of many mistakes I have made. He put in immeasurable efforts to train me to be a researcher and scientist. He sets high standards for all of our work, exhibits inspiring enthusiasm for research, and works very hard to strike for excellence. I greatly appreciate everything he has done for me. He is and will always be a great model for me to pursue excellence in research.

I also thank the other members of my dissertation committee, Prof. Prem Devanbu, Prof. Ron Olsson, and Dr. Dan Quinlan. Their practical, experienced perspective on Software Engineering problems helped improve the work presented in this dissertation. I truely appreciate their efforts on reviewing and perfecting my dissertation under an extremely tight schedule. Although of very different styles from Prof. Su, their passion, dedication, and professionalism are also excellent sources for me to continuously learn from.

I thank my labmates and collaborators who provided invaluable supports not just for research, but also for everyday life. Stoney Jackson, Matt Roper, Nija Shi, Chad Sterling, Brian Toone, Gary Wassermann, and Eric Wohlstadter introduced me to the Kemper 2249 lab. Gary and Nija provided special advices for life as graduate students. Mark Gabel, Stéphane Glondu, Ghassan Misherghi, and Andreas Sæbjørnsen worked hard to support much of our research projects. Section 2.4 is more-so Mark's work than my own, but is included in the dissertation for completeness. Earl Barr, Chris Bird, Jed Crandall, Zhongxian Gu, Taeho Kwan, Sophia Sun, Jeff Wu, and Dennis Xu also frequently gave very useful feedback on ideas, paper drafts, and practice talks to help improve the quality of our work.

The open, collegial atmosphere in the Computer Science department benefited me in many different ways. Quite a few faculty members imparted me much valuable knowledge and advices. The systems and administrative staff paid quite an effort to solve technical and logistical issues for me. I thank them all and the wonderful environment they have created for us students.

My most heartfelt acknowledgement goes to my wife, Xiaojing, and our parents for their constant, unconditional supports during the six years of my Ph.D. study. Their love, encouragements, and comforts are my endless sources of energy and happiness. I dedicate my dissertation and love to them.

Chapter 1

Introduction

Software systems are becoming more and more sophisticated, complex, and ubiquitous, while standards for software quality and productivity are becoming higher and higher. Developers constantly seek various techniques, tools, and practices to speed up software development without introducing additional software defects. One commonly used tactic for this purpose is to reuse prior knowledge existing in previous software projects. Source code is one of the most directly reusable forms of prior knowledge, along with specifications, design documents, and test suites used for previous projects. Developers often copy and paste code to quickly implement functionalities that have been implemented before. This practice is not just applied within the same project, it is also often used across projects, *e.g.*, by embedding one project inside another. The abundance of open source software also facilitates this practice since developers can easily access the code bases of previous open source projects and incorporate portions of that code into their own code bases. According to various studies [92, 101, 104, 119], a large program may have more than 20% of its code similar to some other parts of the program.

The excessive amount of similar code imposes significant cost on and challenges to software management and maintenance. For example, after a piece of code has been copied and pasted to different places within or outside a project, the different instances of the

same piece of code need to be maintained separated; a bug fix in one place may need to be propagated to other places to improve software quality, while consistent propagation of code changes among different instances can be difficult. Manually tracking and propagating code changes across all instances of a copied piece of code can be error-prone and tedious. Studies have shown that the number of software errors and the cost for software maintenance are closely related to the number of lines of code. Jones [99] estimated that there are, on average, five defects in software requirements, design, and code per function point; Humphrey [85] found that even experienced software developers normally inject 100 or more defects per thousand lines of code. The amount of software is huge; more than 250 billions lines of code have being maintained and the number is constantly increasing [164]. Studies have estimated that the relative cost for maintaining software and managing its evolution could represent more than 90% of its total cost, which has been named as *legacy crisis* by Seacord *et al.* [160]. A large number of software defects have occurred, many of which have caused critical incidents, especially when they are found after release. National Institute of Standards and Technology (NIST) has shown that software errors cost the U.S. economy an estimated $59.5 billion annually, or about 0.6% of the gross domestic product [132]. Considering that similar code may account for more than 20% of the sizes of software projects, a significant portion of software maintenance and failure cost may be attributed to similar, redundant code which demands extra testing, debugging, and maintenance effort.

Reducing maintenance cost is thus a strong motivation for detecting similar code in large projects, refactoring them to reduce redundancy and improve software quality, and tracking and managing their migration and evolution.

In addition, detecting, tracking, and managing similar code may also help improve software productivity, enabling faster code development by providing effective ways for developers to utilize prior knowledge embedded in existing code bases. Based on the frequency at which a piece of code is used, a large, searchable code library containing commonly used code can be constructed; the pieces of code in the library can have various granularities and

modalities, from sequences of primary code statements to sets of functions and modules, from co-existent function calls to programming rules, design patterns and even software architectures. When developers can reuse the code library for development of new software in an easy, error-proof way, instead of copying, pasting, and inlining code, new software can then be developed faster with fewer defects since the reused parts are built on the top of previous proven code bases and developers can be freed from many coding details and focus more on overall design issues and new features needed during the development. Just like the Standard Template Library (STL) provides generic algorithms and data structures for C++ programs, and like the Java Class Library in the Java Platform provides much of the same reusable functions commonly found in modern operating systems, the library of commonly used similar code can provide additional reusable code. Different from existing libraries though, such a library can provide not only more reusable code, but also coding patterns and examples showing ways to use the reusable code. For example, the library could contain a set of commonly used APIs and a set of code templates that illustrate the ways those APIs should be used (*e.g.*, the functions `allocate`, `initialize`, `traverse`, `calculate`, and `release` should be called in sequence). Such a library comprised of similar code of different granularities and modalities may help to alleviate the problem of API jungles [126] and enable a new, more efficient way of code reuse.

1.1 Definitions for Similar Code

There have been many studies that define similar code differently. In software engineering, similar code is also known as *cloned code* (or *code clones*), reflecting the fact that much similar code occurs due to the practice of copying and pasting during software development process. Code clones introduced by copying and pasting are mostly of similar appearance, *i.e.*, lexically or syntactically similar, even after developers make changes to the pasted versions. Such code clones may also have similar functionality (*e.g.*, producing the same output given the same input) since they are derived from the same origin. Further, there

are other software engineering practices that may introduce different kinds of code clones. For example, n-version programming [38, 107, 121, 131] produces functionally similar code that is quite different in syntax and even architectural design.

Beside the origins of code clones, applications with different purposes in mind may also affect the definitions of code clones. For example, if code refactoring is carried out mainly based on syntactic similarity among code, a clone definition based on code syntax should be sufficient. If the refactoring is carried out based on functionality regardless of the time and space complexities, *e.g.*, replacing all code that sorts a sequence of data with a call to a common sorting routine, we will need a similarity definition based on code functionality or semantics. For another example, if our goal is to detect code plagiarism among students, we cannot use a purely functionality-based definition since students usually have the same coding task to work on and their code should have the functionality when implemented correctly; if we want to detect code plagiarism in the wild, we will need to consider a similarity definition based on a combination of syntax, functionality, and runtime behavior, not only at source code level but also at binary code level, since plagiarized code may often be obfuscated to avoid detection even though they have the same functionality.

Based on the semantic-awareness of the definitions for code clones, we can have a wide-spectrum, from the least semantic-aware definitions to the most semantic-aware ones. We say a definition is the least semantic-aware when it considers two pieces of code as code clones only if they are exactly the same as character by character, including spaces, tabulators, line breaks, *etc.* Exact string matching algorithms (*e.g.*, KMP [46]) can be applied to find such clones. Such a definition is very restrictive and does not allow any code editing when a piece of code is copied and pasted. More flexible clone definitions have been introduced to tolerate different kinds of code edits. The literature has provided a classification of four types of clones [22, 152], to allow code clones that can be transformed from one to another using different kinds of code editing operations:

Type 1: Code pieces are exactly the same as each other except for differences in white-

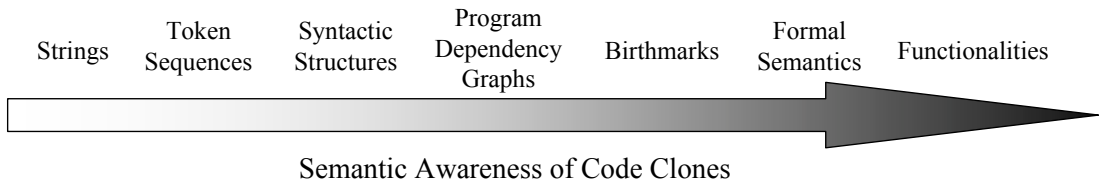


Figure 1.1: From the least semantic-aware definitions to the most semantic-aware ones.

space characters and code comments.

Type 2: Code pieces are syntactically identical except for differences in identifiers, literals, and types.

Type 3: Code pieces are syntactically similar and can be transformed to each other by further edit operations (*e.g.*, statement addition, deletion, and modification).

Type 4: Code pieces have similar semantics but different syntactic appearances.

This classification is constructed mainly according to the possible edit operations performed by developers when they copy and paste a piece of code. It is suitable to classify clones introduced by this coding practice. However, the classification is very coarse-grained; it does not distinguish different kinds of semantic clones (**Type 4**). In this dissertation, we provide a spectrum of code clone definitions based on their semantic-awareness that extends the above classification. Figure 1.1 illustrates the spectrum.

The least semantic-aware definition of code clones treats each program as a sequence of characters or strings and is concerned with the differences among strings, while token-based definitions treat each program as a sequence of tokens and ignore differences due to white-spaces or code comments. Sometimes, differences among certain tokens which are considered non-essential for a particular application may also be ignored. These definitions are usually unable to deal with additional edits or systematically take syntactic boundaries into consideration.

In order to be less sensitive to code formats or names used in programs, syntax-based clone definitions, either based on parse trees (PTs) or abstract syntax trees (ASTs), are

introduced to abstract away many coding details, but preserve essential syntactic structures and consider similarities among such structures during clone detection and analysis.

Further, each code clone may not necessarily be a consecutive segment of its containing program. Code performing one functionality may be interwoven with code performing another functionality. For example, many programs have logging capabilities and the code responsible for logging is often interwoven with the code performing the main functionality. Although the logging code in different programs is seldom consecutive, it is often very similar to each other if we examine it separately from the containing programs. Such situations are getting more attention with the emergence of aspect-oriented software development (AOSD); mining these cross-cutting functionalities or aspects in legacy programs requires different definitions of clones. Some definitions aggregate interdependent code together while separating unrelated code based on program dependencies (data dependencies and control dependencies) among expressions and statements involved, and define similarities among the aggregated code so that non-consecutive code clones can be reduced to consecutive code clones.

There are other semantic-aware definitions than program dependency and aspect-oriented definitions. Birthmark-based ones construct code *fingerprints* that are representative characteristics, either static or dynamic, of each piece of code, and define similarities among such fingerprints. Many kinds of information about the code, *e.g.*, how many execution paths are possible, which functions are called, how the status of a particular variable changes along an execution, what is the behavior of the code when it errs, *etc.*, can be encoded in the fingerprints.

Definitions based on formal semantics, *e.g.*, operational semantics, also exist. They consider pieces of code as clones when they have equivalent formal semantics. Such definitions commonly originated from the concept of program equivalence; they may consider every intermediate state of each piece of code and be too computationally expensive for practical usages.

Functionality-based definitions consider inputs and outputs for each piece of code, and

define code fragments to be clones if outputs from different pieces of code for the same input are similar. Such definitions are not concerned with code appearance, structure, dependencies, or intermediate states, but focus on externally observable behavior of each piece of code, which allows very different implementations of the same functionality. We say such definitions are the most semantic-aware ones for code clones.

No clear cut boundary delineates the many different code clone definitions; actual clone definitions can be a combination of several kinds of definitions in the spectrum. For example, a program dependency based definition can take either syntactic structures or code functionality or both into consideration when defining what are treated as clones. Which definition is appropriate could depend on an actual application of code clone detection and analysis.

1.2 Limitations of Previous Work

Many previous studies have tackled each kind of code clones defined in the spectrum. Different clone detection algorithms have thus been proposed for detecting and analyzing different kinds of clones. Many of them indeed span several kinds of clones in the spectrum.

File comparison tools (*e.g.*, diff and cmp [66]) are commonly available on Unix-like systems, providing byte-based and line-based file comparison. These tools are very primitive for code clone detection purposes. Much work has been carried out to provide tools directly for clone detection and analysis. Baker's work [9–11] is most representative among string-based clones. She has proposed "parameterized" string matching algorithms to find duplicated code in programs, where identifiers and literals are replaced with a global constant.

Token-based definitions are more robust against code changes such as spacing and formatting. Many tools have been developed based on such definitions. CCFinder [101, 124] and CP-Miner [119] are perhaps the most well-known ones among token-based techniques. Such tools usually process programs first to produce token sequences, which are then

scanned for duplicated subsequences that indicate potential code clones. Although such techniques can be very efficient, they often cannot systematically recognize valid syntactic boundaries in programs and cause inappropriate results since clones detected by such techniques may be syntactically invalid units.

Syntactic structure-based definitions are even more robust against more kinds of code changes than token-based ones. Detection techniques based on such definitions usually parse programs into parse trees or abstract syntax trees, then identify exact or close matches of subtrees as clones [18, 19, 174]. Alternatively, various metrics are used to *fingerprint* the subtrees, and subtrees with similar fingerprints are reported as possible code clones [111, 127]. Although these previous tree-based techniques and tools often generate more accurate clone reports than string or token based tools, they are still commonly considered not semantic-aware and do not scale to billions or millions of lines of code.

Studies have also considered more semantic-aware definitions for code clones as shown in the spectrum. First, program dependency graphs (PDGs) [62, 83] are used to represent certain semantic information as data dependencies and control dependencies, which are then searched for similar subgraphs [110, 114, 123]. These previous techniques directly or indirectly rely on graph or subgraph isomorphism which are NP-hard problems [7, 69] and thus do not scale.

Birthmark-based definitions often define particular fingerprints for each piece of code then define code similarities based on the similarities among corresponding code fingerprints. Various kinds of fingerprints have been proposed for detecting illegal theft code or code clones [44, 45, 86, 157, 191]. Although these techniques are often semantic-aware, they are sensitive to the defined fingerprints as well, and which fingerprint is more appropriate may depend on a particular application.

Formal semantic based definitions are closely related to program equivalence [47], which is a classic problem and is undecidable in general. Equivalence based on operational semantics has been proposed long time ago [143, 150]. Due to their computational complexity, such concepts are used mainly for checking whether two given pieces of code are equivalent

or not, instead of searching for unknown code clones in large programs.

Functionality based definitions can be treated as special cases of semantic based definitions where intermediate programs states are ignored and only input and output states for each piece of code are considered for comparing similarity. Such definitions have also been investigated [23, 48, 187], but not been applied in clone detection and analysis. Similar to syntactic structure based techniques, the literature has not shown scalable realizations of these semantic-aware definitions.

1.3 Main Contributions

This dissertation aims to develop scalable, accurate, and practical clone detection algorithms that are based on syntax trees, dependency graphs, and code functionality, and are more semantic-aware and robust against code modifications than previous string, token, or tree based techniques. Also, with the development of practical detection algorithms, this dissertation explores various applications of code clones besides the traditional application of code refactoring. Specifically, the dissertation makes the following contributions:

- It presents a spectrum of definitions of code clones based on semantic-awareness (Figure 1.1) which provides a mean to understand, classify, and relate numerous clone-related studies.

- It presents DECKARD, a scalable and accurate tree-based code clone detection technique and tool. The key insight of DECKARD is to represent syntax trees or dependency graphs of a program as structure-preserving characteristic vectors in the Euclidean space and employ efficient hashing algorithms to cluster these vectors. Experiments have shown that DECKARD scales to millions of lines of code with few false positives. Also, DECKARD is language-agnostic, applicable to any language with a formally specified grammar. In addition, the algorithms within DECKARD can be easily parallelized, making DECKARD suitable for distributed clone detection on the

billions of lines of open source code.

- It proposes a novel application of clone detection to bug detection. In particular, it presents a general notion of context-based inconsistencies as strong indicators of clone-related bugs and applies DECKARD to identify such inconsistencies. Many previously unknown bugs in large projects, *e.g.*, the Linux kernel and Eclipse, are discovered. These bugs exhibit diverse characteristics and are difficult to detect with any single previous bug detection technique.

- It presents EQMINER, the first scalable technique to detect functionally equivalent code for understanding code duplication at the functionality level. Inspired by Schwartz's randomized polynomial identity testing, EQMINER adapts automated random testing to quickly determine the functional equivalence among arbitrary code fragments automatically extracted from a large program. Evaluated on the Linux kernel, EQMINER discovered many functionally equivalent code fragments that are syntactically different.

1.4 Dissertation Outline

The rest of this dissertation is organized according to the aforementioned main contributions. Chapter 2 presents the design of, implementation of, and experiments on DECKARD. It will also show that the general framework proposed in DECKARD—structure vectorization and efficient hashing—can be applicable for clone detection beyond tree-based definitions. Chapter 3 shows a novel application of clone detection to bug detection. It illustrates that clone-related inconsistencies can be a valuable bug finding technique and complement other bug detection techniques. Chapter 4 presents the first scalable functional code clone detection technique applicable to large programs; it not only validates the folklore that functionally equivalent but syntactically different code commonly exist, but also proposes many techniques to address challenges associated with large-scale functional code clone

detection. Then, Chapter 5 discusses more related work that is not necessarily limited to code clone detection and analysis, to broaden the connection of this dissertation to other fields. Finally, Chapter 6 concludes and describes some considerations on future directions for clone detection and analysis.

Chapter 2

Scalable and Accurate Tree-based Clone Detection

Chapter 1 has argued that code clones are common and have many important software engineering applications. Many previous studies have developed various techniques and tools for clone detection. Among previous techniques, CCFinder [101] and CP-Miner [119] (which are token-based clone detection tools), and CloneDR [18,19] (which is a syntax tree-based clone detection tool) represent the state-of-the-art. However, they either do not scale to large code bases due to expensive algorithms for tree-comparison or are not robust against minor code modifications due to their token-based nature. This chapter presents an efficient algorithm for identifying similar subtrees and applies the algorithm to tree representations of source code. The algorithm is based on a novel characterization of subtrees with numerical vectors in the Euclidean space $\mathbb{R}^n$ and an efficient hashing algorithm to cluster these vectors *w.r.t.* the Euclidean distance metric. Subtrees with their corresponding vectors in one cluster are considered similar, and thus indicate code associated with the subtrees are clones. The algorithm is also language independent, applicable to syntax trees generated from any programming language.

This chapter also presents an implementation of the tree similarity algorithm in a tool

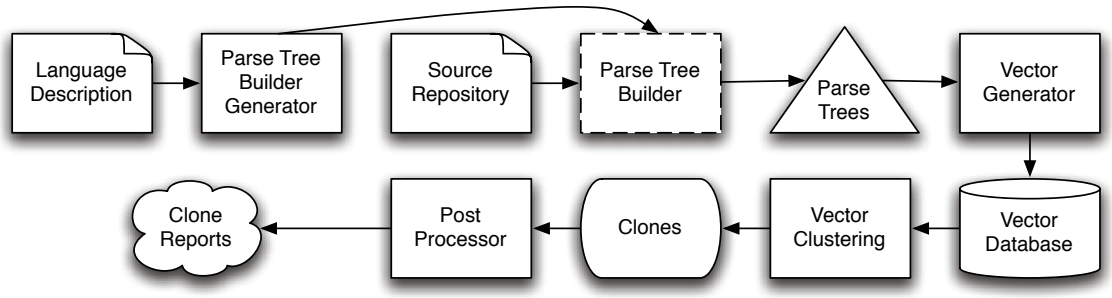


Figure 2.1: DECKARD's system architecture.

called DECKARD. We have done extensive empirical evaluation of DECKARD on large software (including JDK which is written in Java and the Linux kernel which is written in C) and compared DECKARD against CloneDR and CP-Miner. Results indicate that DECKARD is both scalable and accurate: it detects more clones in large code bases than both CloneDR and CP-Miner; it is more scalable than CloneDR, which is also tree-based, and is as scalable as the token-based CP-Miner.

2.1 Overview

As introduced in the beginning of this chapter, the main idea of our algorithm is to compute particular *characteristic vectors* to approximate structural information within trees and then adapt *Locality Sensitive Hashing* (LSH) [50] to efficiently cluster similar vectors (and thus code clones).

Figure 2.1 shows the high level architecture of DECKARD:

- A parser is automatically generated from a formal grammar of a language.
- The parser translates sources files in a program or a set of programs into parse trees.
- The parse trees are processed to produce a set of characteristic vectors whose dimensions are fixed for a particular programming language and capture the syntactic information of parse trees.

- The vectors are clustered by special hashing algorithms; close (*w.r.t.* Euclidean distances) ones are put into the same cluster.
- Additional post-processing heuristics, such as filtering out clusters with tiny or overlapping code fragments, are used to generate code clone reports.

This section aims to illustrate the above main steps of the algorithm with a small example using the following two C program fragments which are array initialization:

```
for (int i= 0; i < n; i++)
    x[i]= 0;
```

```
for (int i= 0; i < n; i++)
    y[i]= "";
```

The parse trees generated by DECKARD for these two code segments are identical, because the code differs only in identifier names and literal values and DECKARD ignores such token-level differences. The parse tree is shown in Figure 2.2. A pairwise (sub)tree comparison could be used to detect such clones, but this is expensive for large programs because of the possibly large number of subtrees. In the following, we demonstrate a novel, efficient technique for tree similarity detection.

Characteristic Vectors We introduce *characteristic vectors* to capture structural information of trees (and forests). This is a key step in our algorithm. The characteristic vector of a subtree is a point $\langle c_1, \ldots, c_n \rangle$ in the Euclidean space, where each c_i represents the count of occurrences of a specific tree pattern in the subtree. For this example, we let the tree patterns be the node kinds in a parse tree and node kinds are basically non-terminals and terminals defined within the grammar of a programming language. We will introduce more general tree patterns in Section 2.2.2.

Not all nodes in parse trees are essential for capturing tree structural information; many are redundant *w.r.t.* their parents, or were introduced to simplify the language grammar specification. We thus also distinguish between *relevant* and *irrelevant nodes*. Example irrelevant nodes include C tokens '[' and ']' used in array expressions and parentheses ('('

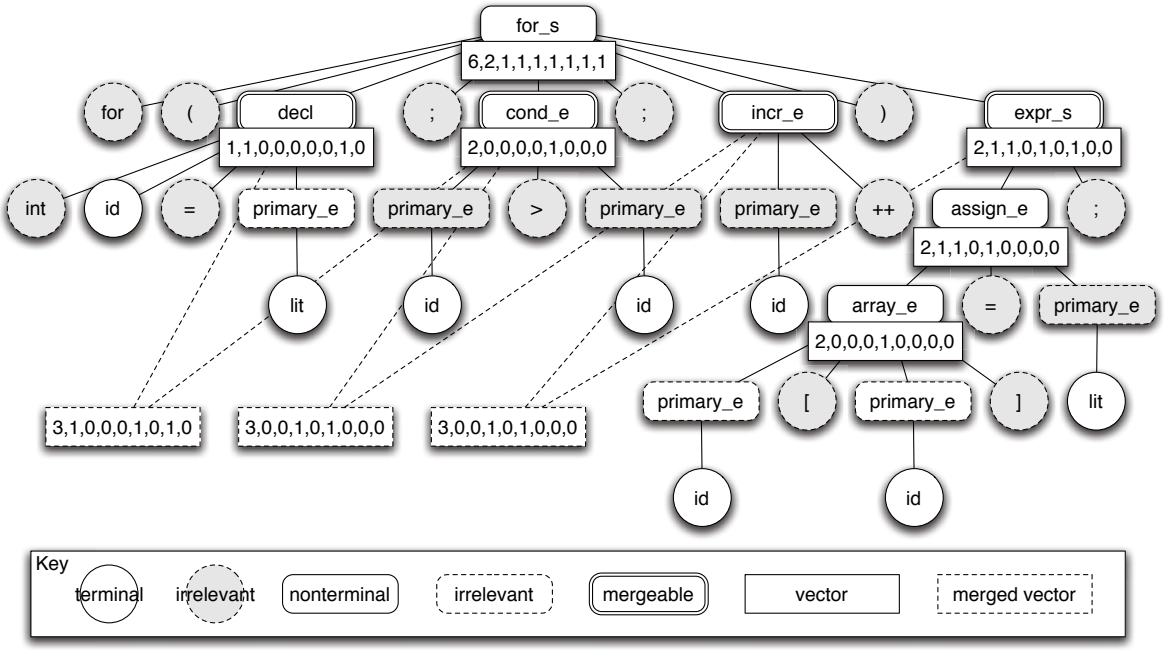


Figure 2.2: A sample parse tree with generated characteristic vectors.

and ‘)’) which may be used in C for exressing explicit precedence, function application, and delimiting conditional clauses.

In Figure 2.2, nodes with solid outlines are relevant while nodes with dotted outlines are irrelevant. Irrelevant nodes do not have an associated pattern or dimension in our vectors. For the example, the ordered dimensions of characteristic vectors are occurrence counts of the relevant nodes: id, lit, assign_e, incr_e, array_e, cond_e, expr_s, decl, and for_s. Thus, the characteristic vector for the subtree rooted at decl is $\langle 1, 1, 0, 0, 0, 0, 0, 1, 0 \rangle$ because there is an id node, a lit node, and a decl node.

Characteristic vectors are generated with a post-order traversal of the parse tree by summing up the vectors for children with the vector for the parent's node. As an example, the vector for the subtree rooted at assign_e $\langle 2, 1, 1, 0, 1, 0, 0, 0, 0 \rangle$ is the sum of the vectors for array_e $(\langle 2, 0, 0, 0, 1, 0, 0, 0, 0 \rangle)$, = $(\langle 0, 0, 0, 0, 0, 0, 0, 0, 0 \rangle)$, primary_e $(\langle 0, 1, 0, 0, 0, 0, 0, 0, 0 \rangle)$, and the additional node assign_e $(\langle 0, 0, 1, 0, 0, 0, 0, 0, 0 \rangle)$. Users may also specify a minimum token count to suppress vectors for small subtrees; this helps to avoid reporting small clones

which are often uninteresting. For example, in Figure 2.2, with this threshold set to three, no vector is generated for the subtree rooted at incr_e. By varying this threshold, we can systematically find only large clones.

Vector Merging The aforementioned technique considers only those code fragments with a corresponding subtree in the parse tree. However, developers often insert code fragments within some larger context. Differences in the surrounding nodes may prevent the parents from being detected as clones (*cf.* Section 2.3.3 for a concrete example from JDK 1.4.2). To identify these cloned fragments, we use a second phase of characteristic vector generation, called *vector merging*, to sum up the vectors of certain node sequences. In this phase, a sliding window moves along a serialized form of the parse tree. The windows are chosen so that a *merged vector* contains a large enough code fragment. In Figure 2.2, for example, we merged the vectors for decl and cond_e to get the vector $\langle 3, 1, 0, 0, 0, 1, 0, 1, 0 \rangle$ for the combined code fragment.

The choice of which nodes in the tree to merge affects which and how many code fragments from source files will be considered as candidates for code clones. Merged nodes should correspond to code fragments that are valid syntactic units in the source, while not frequently containing large subtrees. Roots of expression trees, likely to be units for copy-pasting, are often good choices for merging vectors. We call such chosen nodes *mergeable* nodes. In Figure 2.2, the four children of the **for** statement are used as mergeable nodes. It is not necessary for mergeable nodes to be on the same level. If we had chosen any statement to be mergeable, the entire **for** loop would have been considered as one unit without subsequences. In Figure 2.2, we also required each merged fragment to contain at least five tokens. If we had required six tokens instead, there would have been only two merged vectors instead of three: one for decl and cond_e, and the other for cond_e, incr_e, and expr_s.

Vector Clustering and Post-Processing After we have selected the characteristic vectors, our algorithm clusters similar characteristic vectors *w.r.t.* their Euclidean distances to detect cloned code. The two sample C code fragments both have the same characteristic vector $\langle 6, 2, 1, 1, 1, 1, 1, 1, 1 \rangle$, and DECKARD reports them as clones. Because the number of generated vectors can be large, an efficient clustering algorithm is needed. We will present such an algorithm in Section 2.2.

The subtree rooted at expr_s also illustrates the need for post-processing. When a particular subtree has a low branching factor, the vectors for a child and its parent are usually very similar and thus likely to be detected as clones. We employ a post-processing phase following clustering to filter such spurious overlapping clones.

2.2 Algorithm Description

This section gives a detailed description of our tree similarity algorithm: it first formally defines a *clone pair* (Section 2.2.1), then introduces characteristic vectors for trees and describes how to generate them (Section 2.2.2), and finally explains our vector clustering algorithm for clone detection (Section 2.2.3).

2.2.1 Formal Definitions

In this dissertation, we view clones as syntactically similar code fragments. Thus, it is natural to define the notion of similar trees first. We follow the standard definition and use tree editing distance as the measure for tree similarity.

Definition 2.1 (Editing Distance)**.** The *editing distance* of two trees T_1 and T_2, denoted by $\delta(T_1, T_2)$, is the *minimal sequence* of edit operations (either relabel a node, insert a node, or delete a node) that transforms T_1 to T_2.

Definition 2.2 (Tree Similarity)**.** Two trees T_1 and T_2 are σ*-similar* for a given threshold σ, if $\delta(T_1, T_2) < \sigma$.

We are now ready to define the notion of a *clone pair*.

Definition 2.3 (Clone Pair)**.** Two code fragments C_1 and C_2 are called a *clone pair* if their corresponding tree representations T_1 and T_2 are σ-similar for a specified σ.

Such a definition based on tree editing distance faithfully captures how similar two code fragments are. However, it does not lead naturally to an efficient algorithm due to several reasons:

- The complexity of computing the editing distance between two trees is expensive.[1]
- It requires many pairwise comparisons to locate similar code in large software (quadratic in the worst case).

Instead, we approximate tree structures using numerical vectors and reduce the tree similarity problem to detecting similar vectors. Before describing the details, we define the two common distance measures for numerical vectors that we use in this dissertation.

Definition 2.4 (Distance Measures on Vectors)**.** Let $v_1 = \langle x_1, \ldots, x_n \rangle$ and $v_2 = \langle y_1, \ldots, y_n \rangle$ be two n-dimensional vectors. The *Manhattan distance* of v_1 and v_2, $\mathcal{H}(v_1, v_2)$, is their l_1 norm, *i.e.*, $\mathcal{H}(v_1, v_2) = ||v_1 - v_2||_1 = \sum_{i=1}^{n} |x_i - y_i|$; it is also equivalent to *Hamming distance* if the vectors are binary (whose elements are 1 or 0). The *Euclidean distance* of v_1 and v_2, $\mathcal{D}(v_1, v_2)$, is their l_2 norm, *i.e.*, $\mathcal{D}(v_1, v_2) = ||v_1 - v_2||_2 = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$.

Such distance measures are much easier to compute and efficient algorithms for near-neighbor queries exist for numerical vectors. Based on these observations, we show how to abstract trees into vectors and how to efficiently cluster similar vectors to detect code clones.

2.2.2 Characteristic Vectors for Trees

Recall that in Section 2.1 we illustrated the use of occurrence counts of relevant nodes to abstract a subtree (or subtrees). That example shows a special case of the general

[1] More precisely, for two trees T_1 and T_2 the complexity is $O(|T_1| \times |T_2| \times d_1 \times d_2)$, where $|T_i|$ is the size of T_i and d_i is the minimum of the depth of T_i and the number of leaves of T_i [189].

construction that we will introduce in this section. In particular, we describe a general technique to map a tree (or forests) to a numerical vector which characterizes the structure of the given tree. Without loss of generality, we assume trees are binary [108].

Atomic Tree Patterns and Vectors

Given a binary tree, we define a family of *atomic tree patterns* to capture structural information of a tree. They are parametrized by a parameter q, the height of the patterns.

Definition 2.5 (q-Level Atomic Tree Patterns)**.** A *q-level atomic pattern* is a complete binary tree of height q. Given a label set $\mathcal{L}$, including the empty label ϵ, there are at most $|\mathcal{L}|^{2^q-1}$ distinct q-level atomic patterns.

Definition 2.6 (q-Level Characteristic Vectors)**.** Given a tree T, its *q-level characteristic vector* (denoted by $v_q(T)$) is $\langle b_1, b_2, \ldots, b_{|\mathcal{L}|^{2^q-1}}\rangle$, where b_i is the number of occurrences of the i-th q-level atomic pattern in T.

For example, in Figure 2.2, we used the relevant nodes as the 1-level atomic patterns and characterized trees with their 1-level characteristic vectors.

Abstracting trees as q-level vectors yields an alternative to the standard tree similarity definition based on editing distance. Our plan is to use Euclidean distance between q-level vectors to approximate the editing distance of the corresponding trees. We adapt a result of Yang *et al.* on computing tree similarity [183] to show that this approximation is accurate.

Theorem 2.7 (Yang *et al.*, Thm. 3.3 [183])**.** For any trees T_1 and T_2 with editing distance $\delta(T_1, T_2) = k$, the l_1 norm of the q-level vectors for T_1 and T_2, $\mathcal{H}(v_q(T_1), v_q(T_2))$, is no more than $(4q-3)k$.

For any two integer vectors v_1 and v_2, $\sqrt{\mathcal{H}(v_1, v_2)} \leq \mathcal{D}(v_1, v_2) \leq \mathcal{H}(v_1, v_2)$. Thus we have the following corollary that relates the tree editing distance of two trees with the Euclidean distance of their q-level vectors.

Corollary 2.8. For any trees T_1 and T_2 with editing distance $\delta(T_1, T_2) = k$, the l_2 norm of the q-level vectors for T_1 and T_2, $\mathcal{D}(v_q(T_1), v_q(T_2))$, is no more than $(4q-3)k$ and no less than the square root of the l_1 norm, *i.e.*,

$$\sqrt{\mathcal{H}(v_q(T_1), v_q(T_2))} \leq \mathcal{D}(v_q(T_1), v_q(T_2)) \leq (4q-3)k.$$

Corollary 2.8 suggests that either $\frac{\mathcal{D}(v_q(T_1),v_q(T_2))}{4q-3}$ or $\frac{\sqrt{\mathcal{H}(v_q(T_1),v_q(T_2))}}{4q-3}$ can be used as a lower bound of the tree editing distance $\delta(T_1, T_2)$. When such a lower bound is larger than a specific threshold σ, T_1 and T_2 cannot be σ-similar and thus not a clone pair for the specified σ. On the other hand, when the lower bound is smaller than σ, $\delta(T_1, T_2)$ is likely to be less than σ too. Hence, we reduce the problem of tree similarity to the problem of detecting similar q-level vectors.

Notice that Definition 2.6, Theorem 2.7, and Corollary 2.8 can be relaxed to work on tree forests (a collection of trees) as well because tree forests can be viewed as a tree by adding an additional root. This is important for dealing with code fragments that do not correspond to a single subtree in the parse tree (*cf.* Section 2.1).

Vector Generation

There are two phases of vector generation: one for subtrees and one for subtree forests (for generating merged vectors). Algorithm 2.1 shows how vectors are generated for subtrees. Given a parse tree T, we essentially perform a post-order traversal of T to generate vectors for its subtrees. Vectors for a subtree are summed up from its constituent subtrees (line 5). Certain tree patterns may not be important for a particular application, so we distinguish between relevant and irrelevant tree patterns (a concept that is similar to and generalizes relevant and irrelevant nodes from Section 2.1). If a pattern rooted at a particular node N is relevant (line 6), we look up its index in the vector space using `IndexOf` (line 7) and update the vector correspondingly (line 8).

We also allow vectors to be generated only for certain subtrees, for example those that

Algorithm 2.1 q-Level Vector Generation

```
1:  function QVG(T : tree, C : configuration): vectors
2:      V ← ∅
3:      Traverse T in post-order
4:      for all node N traversed do
5:          V_N ← Σ_{n ∈ children(N)} V_n
6:          if IsRelevant(N, C) then
7:              id ← IndexOf(N, C)
8:              V_N[id] ← V_N[id] + 1
9:          end if
10:         if IsSignificant(N, C) ∧
11:           ContainsEnoughTokens(V_N, C) then
12:             V ← V ⋃ {V_N}
13:         end if
14:     end for
15:     return V
16: end function
```

are more likely to be units of clones, such as subtrees rooted at declarations, expressions and statements. Users can select those *significant* node kinds to generate q-level vectors (line 10). For example, if array_e in Figure 2.2 had been specified as *insignificant*, no vector would have been generated for it. In addition, we may want to ignore small subtrees that contain too few tokens (*cf.* incr_e in Figure 2.2). Users can define a minimal token requirement on the subtrees, which is enforced with `ContainsEnoughTokens` (line 11).

Algorithm 2.2 shows how vectors are generated for adjacent subtree forests. It serializes the parse tree T in post-order, then moves a *sliding window* along the serialized tree to merge q-level vectors from nodes within the sliding window. Because it is not useful to include every node in the serialized tree, we select certain node kinds (called *mergeable nodes*) as the smallest tree units to be included (to make larger code fragments in the context of clone detection). For example, the significant nodes, decl, cond_e, incr_e, and expr_s in Figure 2.2 are specified as mergeable. Users can specify any suitable node kinds as mergeable for a particular application. If both a parent and a child are mergeable, we exclude the child in the sliding window for the benefit of selecting larger clones. This is implemented by `NextNode` in Algorithm 2.2 (line 9).

Users can also choose the width of the sliding window and how far it moves in each

Algorithm 2.2 Vector Merging for Adjacent Tree Forests

```
1: function WVG(T : tree, C : configuration): vectors
2:     ST ← Serialize(T, C);   V ← ∅
3:     step ← 0;   front ← ST.head
4:     back ← NextNode(ST.head, C)
5:     repeat
6:         V_merged ← Σ_{n∈[front,back]} V_n
7:         while back ≠ ST.tail ⋀
8:           ¬ContainsEnoughTokens(V_merged, C) do
9:             back ← NextNode(back, C)
10:            V_merged ← Σ_{n∈[front,back]} V_n
11:        end while
12:        if RightStep(step, C) then
13:            V ← V ⋃ {V_merged}
14:        end if
15:        front←NextNode(front, C)
16:        step ← step + 1
17:    until front = ST.tail
18:    return V
19: end function
```

step, *i.e.*, its stride. Larger widths allow larger code fragments to be encoded together, and may help detect larger clones, while larger strides reduce the amount of overlapping among tree fragments, and may reduce the number of spurious clones. With sliding windows of different widths, our algorithm can generate vectors for code fragments of different sizes and provide a systematic technique to find similar code of any size.

2.2.3 Vector Clustering

Given a large set of vectors $\mathcal{V}$, quadratic pairwise comparisons are computationally infeasible for similarity detection. Instead, we can hash vectors with respect to the Euclidean distances among them, and then look for similar vectors by only comparing vectors with equal hash values.

Locality Sensitive Hashing (LSH) [50, 70] is precisely what we need. It constructs a special family of hash functions that can hash two similar vectors to the same hash value with arbitrarily high probability and hash two distant vectors to the same hash value with arbitrarily low probability. It also helps efficiently find near-neighbors of a query vector.

In the following, we provide some basic background on LSH, then discuss how it is applied for clone detection.

Locality Sensitive Hashing

Definition 2.9 ((p_1, p_2, r, c)-Sensitive Hashing)**.** A family $\mathcal{F}$ of hash functions $h : \mathcal{V} \to \mathcal{U}$ is called (p_1, p_2, r, c)-*sensitive* ($c \geq 1$), if $\forall v_i, v_j \in \mathcal{V}$,

$$\begin{cases} \text{if } \mathcal{D}(v_i, v_j) < r & \text{then } Prob[h(v_i) = h(v_j)] > p_1 \\ \text{if } \mathcal{D}(v_i, v_j) > cr & \text{then } Prob[h(v_i) = h(v_j)] < p_2 \end{cases}$$

For example, Datar *et al.* have shown that the following family of hash functions, which map vectors to integers, is locality sensitive [50]:

$$\{h_{\alpha,b} : \mathbb{R}^d \to \mathbb{N} \mid h_{\alpha,b}(v) = \lfloor \frac{\alpha \cdot v + b}{w} \rfloor,\ w \in \mathbb{R}, b \in [0, w]\}$$

Definition 2.10 ((r, c)-Approximate Neighbor)**.** Given a vector v, a vector set $\mathcal{V}$, a distance r, and $c \geq 1$, $\mathcal{U} = \{u \in \mathcal{V} \mid \mathcal{D}(v, u) \leq cr\}$ is called an *rcAN* set of v, and any $u \in \mathcal{U}$ is a (r, c)-*approximate neighbor* of v.

Given a vector set $\mathcal{V}$ of size n and a query vector v, LSH may establish hash tables for $\mathcal{V}$ and find v's rcAN set in $O(dn^\rho \log n)$ time and $O(n^{\rho+1} + dn)$ space, where d is the dimension of the vectors and $\rho = \log_{p_2} p_1 < \frac{1}{c}$ for $c \in [1, +\infty)$. As long as we feed r (the largest distance allowed between v and its neighbors) and p_1 (the minimal probability that two similar vectors have the same hash value) to LSH, it automatically computes other parameters that would give optimal running time of a query.

LSH-based Clone Detection

LSH's querying functionality can help find every vector's rcAN sets, which are needed for clone detection. Algorithm 2.3 describes the utilization of LSH:

Algorithm 2.3 LSH-based Clone Detection

```
1:  function LSHCD(𝒱 : vectors, r : distance, p1 : prob): rcANs
2:      𝒩 ← ∅;   LSH(𝒱, r, p1)
3:      repeat   pick a v ∈ 𝒱
4:          rcAN ← queryLSH(v)
5:          if |rcAN| > 1 then
6:              𝒩 ← 𝒩 ⋃{rcAN \ ⋃_{n∈𝒩} n}
7:          end if
8:          𝒱 ← 𝒱 \ rcAN
9:      until 𝒱 = ∅
10:     return PostProcessing(𝒩)
11: end function
```

- All vectors are stored into LSH's hash tables (line 2), where r serves as the threshold σ defined in Definition 2.3.
- A vector v is used as a query point to get an rcAN set (lines 3 and 4).
- If the rcAN set only contains v itself, it means v has no neighbors within distance σ and should be deleted directly (line 8).
- Otherwise, the rcAN set is treated as a clone class (lines 6 and 8). Such a process may query LSH n times in the worst case.

Thus, our LSH-based clone detection takes $O(dn^{\rho+1} \log n)$ time, where d is the dimension of the vectors, *i.e.*, $|\mathcal{L}|^{2^q-1}$ in terms of q-level vectors, where $|\mathcal{L}|$ is the number of node kinds in a parse tree.

All the rcAN sets may contain potentially spurious clones (*cf.* Section 2.1) and are post-processed to generate clone reports. A filter is created to examine the line range of every clone in an rcAN set and remove any that is contained by or overlaps with others. A second filter is applied after the first one to remove rcAN sets that contain only one vector. Both filters run in linear time in the number of rcAN sets and quadratic time in the size of the sets.

2.2.4 Size-Sensitive Clone Detection

Definition 2.3 of a clone pair does not take into account the varying sizes of code fragments. It is however natural to allow more edits for larger code fragments to be still considered clone pairs. In this section, we introduce a size-sensitive definition of code clones and an algorithm for detecting such clones. Such a higher tolerance to edits for larger code fragments facilitates the detection of more large clones.

Definition 2.11 (Code Size)**.** The *size* of a code fragment C in a program P, denoted by $S(C)$, is the size of its corresponding tree fragments (subtree forest) in the parse tree of P.

Definition 2.12 (Size-Sensitive Clone Pair)**.** Two code fragments C_1 and C_2 form a *size-sensitive clone pair* if their corresponding trees T_1 and T_2 are $f(\sigma, S(C_1), S(C_2))$-similar, where f is a monotonic, non-decreasing function with respect to σ and $S(C_i)$.

Clone detection based on Definition 2.12 requires larger distance thresholds for larger code. We now present a technique to meet such a requirement. The basic idea is *vector grouping*: vectors for a program are separated into different groups based on the sizes of their corresponding code fragments; then LSH is applied on each group with an appropriate threshold; and finally, all reported clone classes from different groups are combined.

Any grouping strategy is appropriate as long as it meets the following requirements:

- It should not miss any clones detectable with a fixed threshold, thus each group should overlap with the neighboring groups.
- It should not produce many duplicate clones, thus overlapping should be avoided as much as possible.
- It should produce many small groups to help reduce clustering cost.

Algorithm 2.4 shows a generic vector grouping algorithm, where s is a user-specified code size for the first group. Each vector v is dispatched into groups whose size ranges contain the size of its corresponding code fragment, *i.e.*, $S(C_v)$. sizeRanges shows our

Algorithm 2.4 Vector Grouping

```
1: function VG(V : vectors, r : distance, s : size)
2:     R ← sizeRanges(V, r, s)
3:     dispatch V into groups according to the ranges in R
4: end function
5:
6: function SIZERANGES(V : vectors, r : distance, s : size)
7:     The code size range for the 1st group ← [0, s + r]
8:     The range for the 2nd group ←
9:                         r = 0 ? [s + 1, s + 1] : [s, s + 3r + 1]
10:    repeat  compute [l_{i+1}, u_{i+1}] as
11:        l_{i+1} ← r = 0 ? (u_i + 1) : (u_i − (l_i/s) r)
12:        u_{i+1} ← r = 0 ? (u_i + 1) : ((s+2d)/s u_i − 2(d^2/s^2) l_i + 1)
13:    until  u_i ≥ max_{v∈V}{S(C_v)}
14: end function
```

formulae for grouping. The exact constraints used to deduce the grouping formulae can vary as long as they meet the aforementioned requirements.

We can estimate $S(C)$ with the size of C's vector $v = \langle x_1, \ldots, x_n \rangle$, *i.e.*, $S(C) \approx S(v) = \sum_{i=1}^{n} x_i$. Although irrelevant nodes may cause $S(v) < S(C)$, this should have little impact on clone detection because each $S(C)$ is adjusted accordingly.

It is also worth mentioning that vector grouping has the added benefit to improve scalability of our detection algorithm. Because the vectors are separated into smaller groups, the number of vectors will usually not be a bottleneck for LSH, thus enabling the application of LSH on larger programs. In addition, because vector generation works on a file-by-file basis and the separated vectors are processed one group at a time, our algorithm can be easily parallelized. and its performance has a great potential to be improved.

2.3 Implementation and Empirical Evaluation

This section discusses our implementation of DECKARD and presents a detailed empirical evaluation of it against two previous state-of-the-art tools: CloneDR [18, 19] and CP-Miner [119].

2.3.1 Implementation

In our implementation, we use 1-level vectors to capture tree structures. DECKARD is language independent and works on programs in any programming language that has a context-free grammar. It automatically generates a parse tree builder to build parse trees required by our algorithm. DECKARD takes a YACC grammar and generates a parse tree builder by replacing YACC actions in the grammar's production rules with tree building mechanisms. The generated parse tree builders also have high tolerance of syntactic errors. They can skip code portions that they cannot parse, recover from error states, and continue the tree building process from where they can parse. Thus, even if the given YACC grammar for a language does not completely or accurately specify the language, the generated parse tree builders can still be highly usable. For example, the YACC grammar we used for Java does not recognize the `assert` keyword added in Java 1.4, but the generated parse tree builder still successfully built parse trees for 8,451 files out of 8,453 in JDK 1.4.2.

Section 2.3.3 will show that DECKARD works effectively for both C and Java. In addition, YACC grammars are available for many languages, often with the requisite error recovery to localize syntax problems. Thus, it should be straightforward to port DECKARD to other languages. In particular, the ROSE open source compiler infrastructure [146,147,156] supports Fortran, C, C++, Python, and other languages. It can construct unified parse trees and abstract syntax trees for most programs written in the supported languages. It could potentially enable cross-language clone detection and code reuse in the future.

2.3.2 Experimental Setup

We performed extensive experiments on DECKARD, and the most detailed ones were on JDK 1.4.2 (8,534 Java files, 2,418,767 LoC)[2] and Linux kernel 2.6.16 (7,988 C files, 5,287,090 LoC). We also compared DECKARD to both CloneDR [18, 19], a well-known AST-based clone detection tool for Java, and CP-Miner [119], a token-based tool for C, to show that

[2]Our current implementation of DECKARD cannot parse programs written in Java 1.5 or later because the YACC grammar we used for Java does not recognize many new language features (*e.g.*, generics).

DECKARD can achieve higher clone detection rates with comparable computational cost.

To compare with CloneDR, we ran experiments on a workstation with a Xeon 2GHz processor and 1GB of RAM, with both Windows XP (for CloneDR) and Linux kernel 2.4.27 (for DECKARD). CloneDR has several parameters that may affect its clone detection rates, and we chose the most lenient values for all those parameters.

- The minimal depth of a subtree to be considered a clone is set to two.
- The minimal number of tree nodes a clone should contain is set to three.
- The maximal number of parameters allowed when using parameterized macros to refactor clones is set to 65535.
- *Similarity* is set to a value between 0.9 and 1.0, where CloneDR [19] defines *Similarity* as the following:

$$Similarity(T_1, T_2) = \frac{2H}{2H + L + R} \qquad \text{(Eq. 1)}$$

  where H is the number of shared nodes in trees T_1 and T_2, L is the number of different nodes in T_1, and R is the number of different nodes in T_2. This definition takes tree sizes into account, similar to our definition in Section 2.2.4.

To make our comparisons fair despite the different configuration options in each, we compute DECKARD's threshold σ from *Similarity* as follows. Suppose v_1 and v_2 are the 1-level vectors for T_1 and T_2 respectively. Because the l_1 norm of v_1 and v_2 can be approximated as $L + R$ and $l_2 \geq \sqrt{l_1}$ for integer vectors, we can transform a given *Similarity* s to an approximate l_2 distance:

$$\begin{aligned} \mathcal{D}_s(v_1, v_2) &\geq \sqrt{\mathcal{H}(v_1, v_2)} \approx \sqrt{L + R} \\ &\overset{\{\text{Eq. 1}\}}{=} \sqrt{(1-s) \times (|T_1| + |T_2|)} \\ &\geq \sqrt{2(1-s) \times min(S(v_1), S(v_2))} \end{aligned}$$

Given a vector group $\mathcal{V}$, $\sqrt{2(1-s) \times min_{v \in \mathcal{V}} S(v)}$ can serve as the threshold σ used by DECKARD for the group. This is similar to Section 2.2.4, where we use vector sizes to approximate tree sizes. In Figures 2.3, 2.6, 2.4, 2.7, and 2.8, we show *Similarity* only, without showing the derived σ.

To compare with CP-Miner (available for Linux), we ran experiments on a workstation running Linux kernel 2.6.16 with an Intel Xeon 3GHz processor and 2GB of RAM. CP-Miner uses a different distance metric, called *gap*, which is the number of statement insertions, deletions, or modifications to transform one statement sequence to another. Such a parameter is invariant *w.r.t.* different code sizes.

2.3.3 Evaluation Results

We have evaluated DECKARD in terms of the following: clone quantity (*i.e.*, number of detected clones), clone quality (*i.e.*, number of false clones), and its scalability. Our results indicate that DECKARD performs significantly better than both CloneDR and CP-Miner.

Clone Quantity

We measure clone quantity by the number of lines of code that are within detected clone pairs. In the first experiment, we compared DECKARD with CloneDR on JDK. It seems that CloneDR was not able to parse files containing certain keywords (*e.g.*, `assert`). CloneDR also failed to work on the entire JDK at once. Thus, we excluded those syntactically problematic files reported by CloneDR and separated the remaining files into nine overlapping

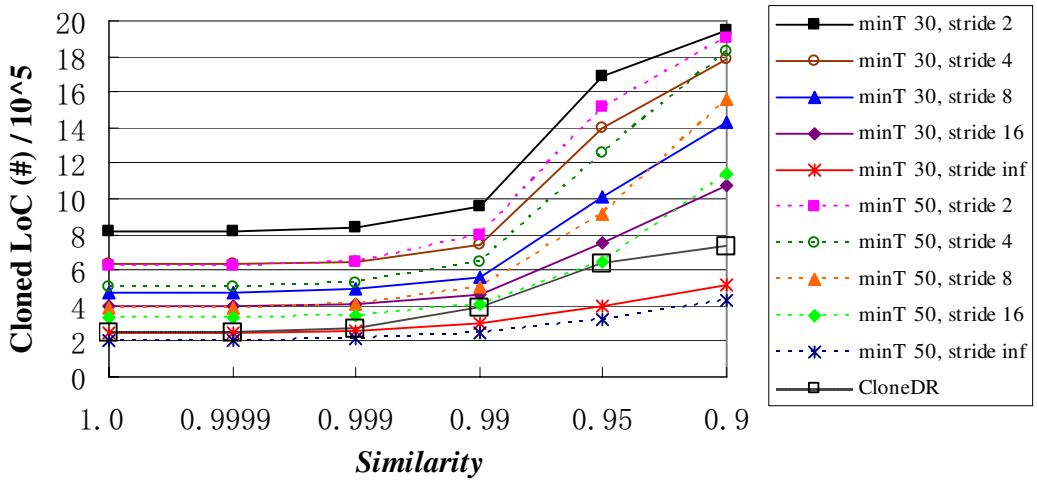


Figure 2.3: Cloned lines of code detected by DECKARD (with grouping and full parameter tuning) and CloneDR on JDK 1.4.2.

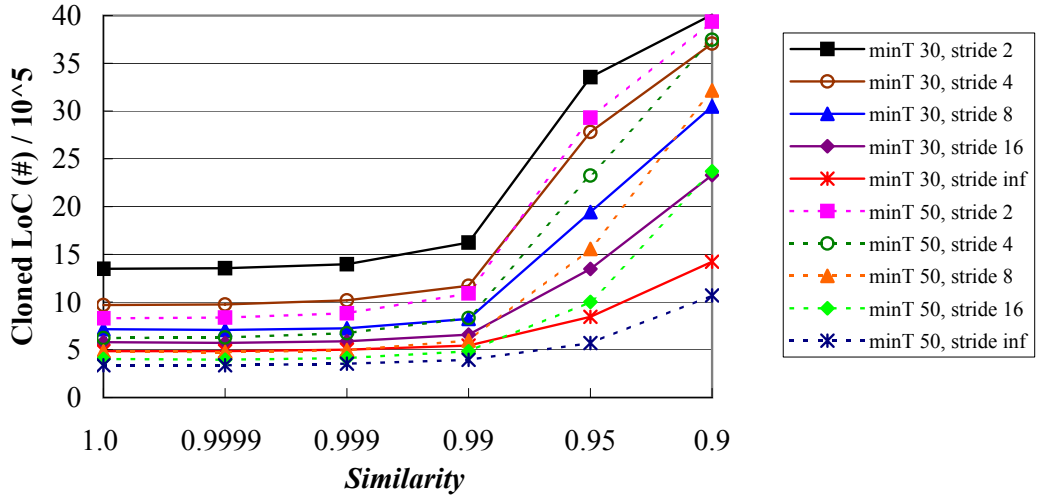


Figure 2.4: Cloned lines of code detected by DECKARD (with grouping and selective parameter tuning) on Linux kernel 2.6.16.

groups, with each group containing around 1,000 files.[3] Figure 2.3 shows the total detected cloned lines over many runs on JDK. For DECKARD, we used a variety of configuration

[3]To reduce potential false negatives for CloneDR, we separated the files into as few groups as possible and nine was the result of a trial-and-error process. We also assume clones most likely occur among spatially close files (*e.g.*, files stored under the same directory), so we listed all file names with their path names from JDK in the dictionary order and then split the ordered list; and files in the same bottom directory were always put into the same group. In addition, we allowed two adjacent (*w.r.t.* the ordered list) groups to share about 100 files to further help reduce potential false negatives.

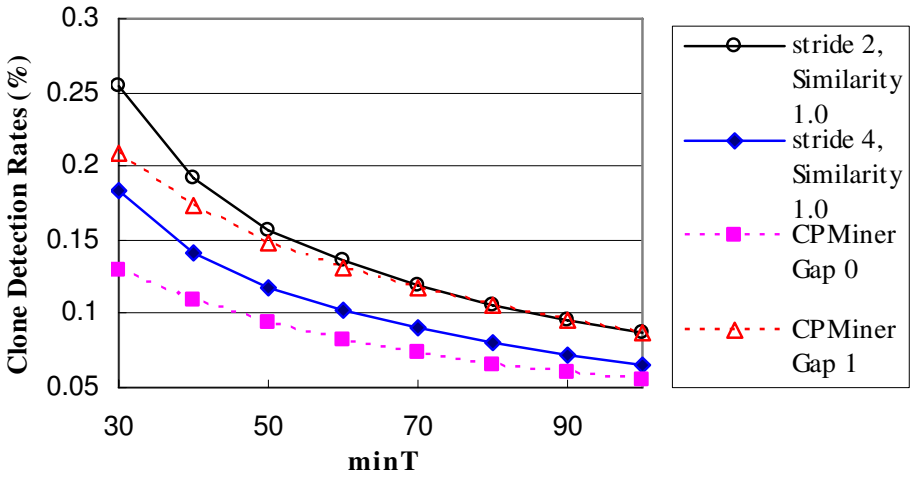


Figure 2.5: Effects of minT on clone detection rates (percentage of cloned lines of code) of DECKARD and CP-Miner on Linux kernel 2.6.16.

options: minT (the minimal number of tokens required for clones) was set to 30 or 50, stride (the size of the sliding window) ranged from 2 to infinity (*i.e.*, $+\infty$, meaning that vector merging is disabled), and *Similarity* ranged between 0.9 and 1.0. The total number of cloned lines detected by DECKARD ranges from 204,263 to 1,943,777, while the number of cloned lines detected by CloneDR ranges from 246,708 to 727,701.

In our second experiment, we compared DECKARD with CP-Miner on the Linux kernel. Figure 2.4 shows the total number of clone lines detected by DECKARD under different configuration options with minT set to 30 or 50, stride ranging from 2 to infinity, and *Similarity* ranging from 0.9 to 1.0. The total number of detected cloned lines ranges from 338,519 to 3,936,242. For CP-Miner, we used four configuration options with minT set to 30 or 50 and gap set to 0 or 1. Its total number of detected clone lines ranges from 498,113 to 1,108,062 as shown in Table 2.1. It failed to operate with $gap > 1$.

In addition, Figure 2.5 plots the decline in clone detection rates as minT increases for both CP-Miner and DECKARD. Even with *Similarity* set to 1.0, DECKARD detects more clones than CP-Miner.

minT	Gap	Cloned LoC (#)	Time (min)
30	0	684,119	18.7
	1	1,108,062	19.7
50	0	498,113	11.9
	1	783,925	18.7

Table 2.1: Cloned lines of code and running time for CP-Miner on Linux kernel 2.6.16.

Clone Quality

The number of reported spurious clones is also important in assessing clone detection tools. We performed random, manual inspection on rcAN sets (*i.e.*, clustered similar vectors) using two criteria:

- Does an rcAN set contain at least one clone pair that corresponds to copy-pasted fragments?
- Are all clones in an rcAN set copies of one another?

If a set fails to satisfy either of the criteria, we classify it as a false clone report. However, it may be difficult to decide for certain whether two code fragments are clones or not. For example, consider the following code fragments from JDK 1.4.2:

```
1  else if (option.equalsIgnoreCase("basic")) {
2          bBasicTraceOn = true;
3  } else if (option.equalsIgnoreCase("net")) {
4          bNetTraceOn = true;
5  } else if (option.equalsIgnoreCase("security")) {
6          bSecurityTraceOn = true;
7  } else ...
8  ...
9  else if (opt.equals("-nohelp")) {
10        nohelp = true;
11 } else if (opt.equals("-splitindex")) {
12        splitindex = true;
13 } else if (opt.equals("-noindex")) {
14        createindex = false;
15 } else ...
```

	CloneDR	CP-Miner	LSH	LSH w/ Grouping
Time	$O(\frac{n^2}{\lvert Buckets\rvert})$	$O(m^2)$	$O(dn^\rho \log n)$	$O(d\sum_{g\in G} \lvert g\rvert^\rho \log \lvert g\rvert)$
Mem	$O(n)$	$O(m)$	$O(n^{\rho+1} + dn)$	$O(max_{g\in G}\{\lvert g\rvert^{\rho+1} + d\lvert g\rvert\})$

	DECKARD w/ Post-Processing
Time	$O(n + d\sum_{g\in G} \lvert g\rvert^{\rho+1} \log \lvert g\rvert + c\lvert rcAN\rvert^2)$
Mem	$max\{O(c\lvert rcAN\rvert), O_{g\in G}(\lvert g\rvert^{\rho+1} + d\lvert g\rvert)\}$

Table 2.2: Worst-case complexities of CloneDR, CP-Miner, and DECKARD (m is the number of lines of code, n is the size of a parse tree, $|Buckets|$ is the number of hash tables used in CloneDR, d is the number of node kinds, $|g|$ is the size of a vector group, $0 < \rho < 1$, c is the number of clone classes reported, and $|rcAN|$ is the average size of the clone classes).

The code between lines 1–7 and that between lines 9–15 have identical structure but different variable names, functions, and constants. CloneDR and CP-Miner may detect them as clones if the two **`if-else`** sequences are standalone statements, but miss them if they are in the middle of different, larger **`if-else`** statements. DECKARD always detects them with reasonably small settings for minT and stride.

We inspected 100 randomly selected rcAN sets reported by DECKARD for JDK 1.4.2 with minT set to 50, stride set to 4, and *Similarity* set to 1.0. Of those, 93 rcAN sets are clearly real clones. Among the remaining seven rcAN sets, three involve **`if-else`** and **`switch-case`** that are similar to the above **`if-else`** example, three involve sequences of simple `import` statements, and one involves sequences of simple declarations. Although it is unclear whether these are clones, the reported clone pairs are all structurally the same. Also because both CloneDR and CP-Miner may detect such code as clones, we also classified these as real clones. This experiment indicates that DECKARD is highly accurate. Because the version of CloneDR that we have does not output the actual clones, we cannot directly compare its accuracy with DECKARD. For future work, we plan to develop a better user interface for DECKARD, which would allow us to conduct further user studies and to more rigorously assess the quality of reported clones.

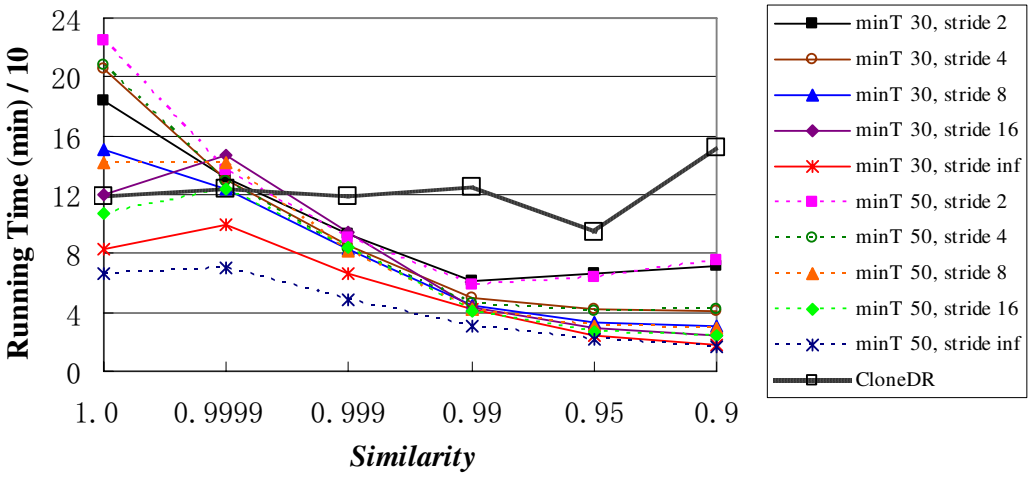


Figure 2.6: Running time of DECKARD (with grouping and full parameter tuning) and CloneDR on JDK 1.4.2.

Scalability

Table 2.2 presents the worst-case time and space complexities for CloneDR, CP-Miner, DECKARD, and LSH. Although the number of tree nodes n is usually several times larger than the number of statements m in a program, DECKARD's performance is still comparable to CP-Miner for large programs because ρ is usually much smaller than one. With vector grouping, LSH's memory consumption can be significantly reduced to make DECKARD scale to very large programs.

Figure 2.6 plots running times for both DECKARD and CloneDR on JDK. We see that DECKARD is several times faster than CloneDR when *Similarity* is less than 0.999. Here, we also show that DECKARD can be configured to run significantly faster without much accuracy loss. By default, LSH takes $O(kd\sum_{g\in G}|g|^{\rho}\log|g|)$ time to tune its own parameters and build optimal (*w.r.t.* query time) hash tables, where k is the number of iterations it uses to find the optimal parameters. Such cost accumulates when the vectors are split into groups, and thus LSH may spend much time on parameter tuning. Reusing the parameters computed for certain groups (*e.g.*, the largest group) can dramatically reduce LSH's running time with little effect on clone quantity and quality. Table 2.3 shows the effectiveness of such

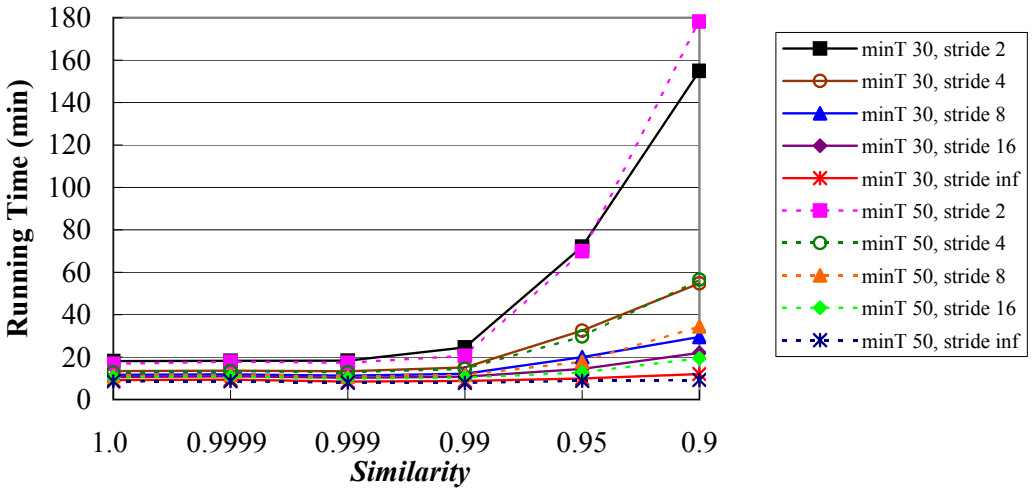


Figure 2.7: Running time for DECKARD (with grouping and selective parameter tuning) and CP-Miner on Linux kernel 2.6.16.

	Sim	**G** (#)	**Cloned LoC** (#)	**T** (min)
Full Tuning	1.0	1984	624265	224.8
Selective Tuning			624265	14.9
Full Tuning	0.99	235	792326	58.6
Selective Tuning			792298	16.3

Table 2.3: Effects of selective parameter tuning in LSH. The data is for JDK 1.4.2, with minT 50, stride 2.

a strategy in reducing the overall running time of DECKARD, especially when the vectors are split into many groups. Figure 2.8, compared with Figure 2.6, gives a more clear view of the benefit of the strategy: DECKARD's running time was 4 to 12 times shorter and became about 6 times faster than CloneDR.

Figure 2.7 shows DECKARD's running time on the Linux kernel with selective parameter tuning. When $Similarity > 0.95$, DECKARD runs in tens of minutes and is comparable to CP-Miner (*cf.* Table 2.1); it can be even faster when *Similarity* is close to 1.0. When $Similarity \leq 0.95$, DECKARD may take more time than CP-Miner. This extra cost is reasonable considering that DECKARD is tree-based and detects more clones, while CP-Miner is token-based and cannot operate with $gap > 1$, and that $Similarity \leq 0.95$ is often too small for clone detection tasks.

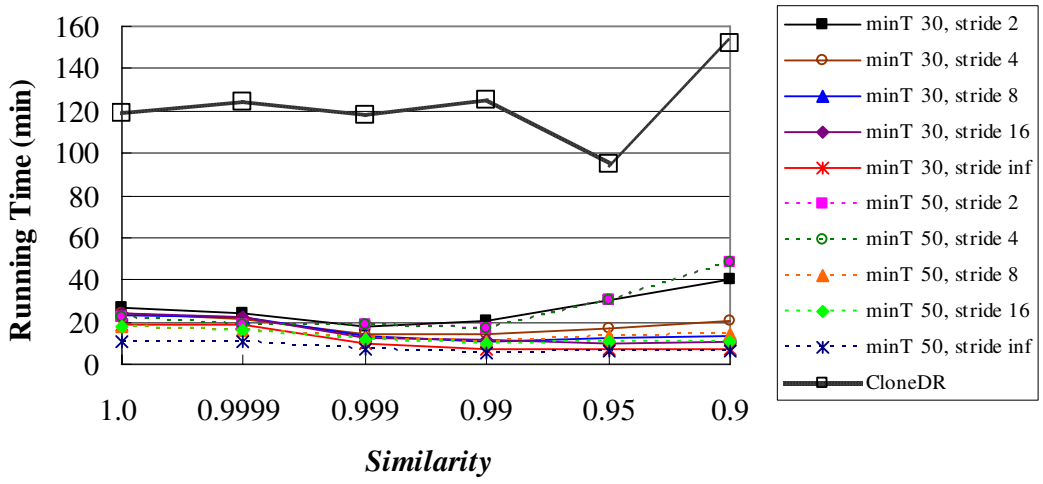


Figure 2.8: Effects of selective parameter tuning in LSH. Comparable minT and stride to Figure 2.6.

2.4 Extending to Graph Based Clone Detection

Code clones detected by DECKARD and other previous tools are mainly limited to contiguous code fragments within a program; they are sensitive to even the most simple structural differences in otherwise semantically similar code, such as inserting unrelated statements inside of a code fragment, replacing a syntactic structure with a semantically equivalent one. As an example, consider the pair of code snippets in Figure 2.9: both perform the same overall computation, but the snippet on the right contains extra statements to time the loop. Since the second code snippet has computationally unrelated code interleaved, previous techniques based on tokens or trees are unable to detect such clones. Techniques have been proposed to use program dependence graphs (PDGs) [62] to find such clones since the computationally unrelated code may become naturally separated from others in PDGs and the problem of interleaved code is less a concern.

Figure 2.10 shows the PDG for the code in Figure 2.9. A PDG is a static representation of data and control dependencies through a procedure. The nodes of a PDG consist of program points constructed from the source code: declarations, simple statements, expressions, and control points. A control point represents a point at which a program branches,

```
1  int func(int i, int j) {
2     int k = 10;
3
4     while (i < k) {
5        i++;
6     }
7
8     j = 2 * k;
9     printf("i=%d, j=%d\n",
10           i, j);
11    return k;
12 }
```

```
1  int func_timed(int i, int j) {
2     int k = 10;
3
4     long start = get_time_millis();
5     long finish;
6     while (i < k) {
7        i++;
8     }
9     finish = get_time_millis();
10    printf("loop took %dms\n",
11           finish - start);
12
13    j = 2 * k;
14    printf("i=%d, j=%d\n", i, j);
15    return k;
16 }
```

Figure 2.9: Example of non-contiguous code clones.

loops, or enters or exits a procedure and is labeled by its associated predicate.

The edges of a PDG encode the data and control dependencies between program points. Given two program points p_1 and p_2, there exists a directed data dependency edge from p_1 to p_2 if and only if the execution of p_2 depends on data calculated directly by p_1. For example, consider the statements on lines 2 and 8 of the code on the left in Figure 2.9. The second statement calculates a value that is initialized in the first. This dependency is illustrated by a directed edge between the two nodes in Figure 2.10. Note that the node corresponding to the formal parameter `j` does not have any outgoing edges. This accurately reflects the fact that `j` is redefined without ever being used at line 8 in the code. The incrementing of `i` on line 5 constitutes both a use and a definition, so the node in the PDG corresponding to `i++` has both a self data dependency loop and outgoing data dependency edges.

Similarly, there exists a directed control dependency edge from p_1 to p_2 if and only if the choice to execute p_2 depends on the test in p_1. The while loop on line 4 of the code illustrates the use of control dependency edges flowing from a control point node. The corresponding PDG node in Figure 2.10 is labeled with the guard expression `i<k`, and

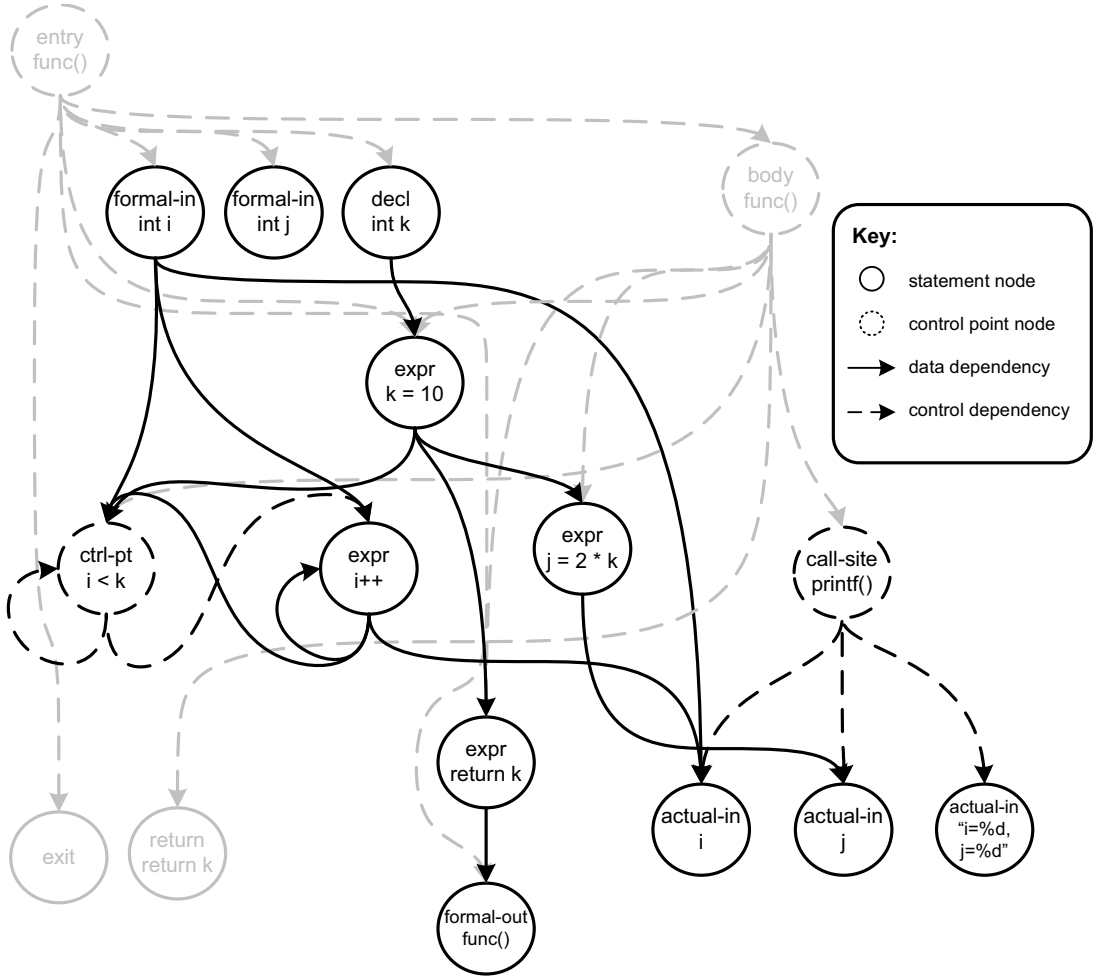


Figure 2.10: The PDG for the code on the left side in Figure 2.9.

there is a control dependency edge to the enclosed increment statement and a self control dependency edge. Function calls are modeled as control points that control the execution of expressions corresponding to the calculation of the actual parameters and the assignment of the return value (or assignments to out parameters). For example, the call to `printf` on line 10 is modeled in this way: there are three outgoing control edges that connect to the three parameters.

PDGs may also contain implicit nodes that do not have a direct source correspondence. These include entry, exit, and function body control points, and are represented by a light shade in Figure 2.10. Since we focus on intraprocedural dependencies, these nodes are

omitted for simplicity.

Previous PDG-based similarity detection tools use various algorithms for subgraph isomorphism, which is a computationally expensive problem, to detect either similar procedures or code fragments [110, 123]. Those techniques have not been shown to scale to million-line programs.

In the following, we present an extended definition of code clones, based on PDG similarity, that captures more semantic, non-contiguous information. We then provide a scalable, approximate algorithm for detecting these clones. We reduce the difficult graph similarity problem to a simpler tree similarity problem by creating a mapping between PDG subgraphs and their related structured syntax.

2.4.1 Definition of PDG-Based Code Clones

Since we already have Definition 2.3 for tree-based code clones and we can assume there is always a mapping function, ρ, that maps a sequence of syntax (of arbitrary type) to a PDG subgraph, we can expand the tree-based definition to graph-based clones.

Definition 2.13 (PDG-Based Semantic Code Clones)**.** Two disjoint, possibly noncontiguous sequences of program syntax C_1 and C_2 are PDG-based semantic code clones if and only if C_1 and C_2 are syntactic code clones or $\rho(C_1)$ is (sub-)graph isomorphic to $\rho(C_2)$.

Applying this relaxed definition, a subset of the code on the right in Figure 2.9 (excluding the code for timing) has exactly the same PDG as the code on the left, and thus the two pieces of code can be considered clones.

2.4.2 Algorithm

The problem is to locate these PDG-based semantic clones in a scalable manner. Previous techniques were not able to scale mainly because of two reasons:

- There is a combinatorial explosion of possible clone candidates to consider.

- Although graph isomorphism testing may be feasible for small, simple PDGs, it is computationally expensive in general. Any method that would require pairwise comparisons would not scale.

We propose a scalable, approximate technique for locating such clones based on the fact that both structured syntax trees and dependence graphs are derived from the original source code and a mapping function ρ can always be constructed between a PDG subgraph and its corresponding syntax. We refer to this associated syntax as the syntactic image, and the algorithm employed in DECKARD can also be applied on syntactic images to help identify PDG-based semantic clones.

Definition 2.14 (Syntactic Image)**.** The syntactic image of a PDG subgraph G, $\mu(G)$, is the maximal set of AST subtrees that correspond to the concrete syntax of the nodes in G. The set is a dominating set, *i.e.*, for all pairs of trees $T, T' \in \mu(G)$, $T \nsubseteq T'$.

Mapping a PDG subgraph to an AST forest effectively reduces the graph similarity problem to an easier tree similarity problem that we can solve efficiently using DECKARD. Based on this idea, the overall algorithm for detecting PDG-based semantic clones are straightforward and is shown in Figure 2.11. At a high level, the algorithm functions as follows:

1. We run DECKARD's primary vector generation. Subtree and sliding window vectors efficiently provide contiguous syntactic clone candidates for the entire program.

2. For each procedure, we enumerate a finite set of *significant subgraphs*; *i.e.*, we enumerate subgraphs that hold semantic relevance and are likely to be good semantic clone candidates. Intuitively, we produce subgraphs of maximal size that are likely to represent distinct computations. Details about selecting such subgraphs are in our paper [67].

3. For each subgraph G, we compute $\mu(G)$ to generate its syntactic image which is essentially an AST forest.

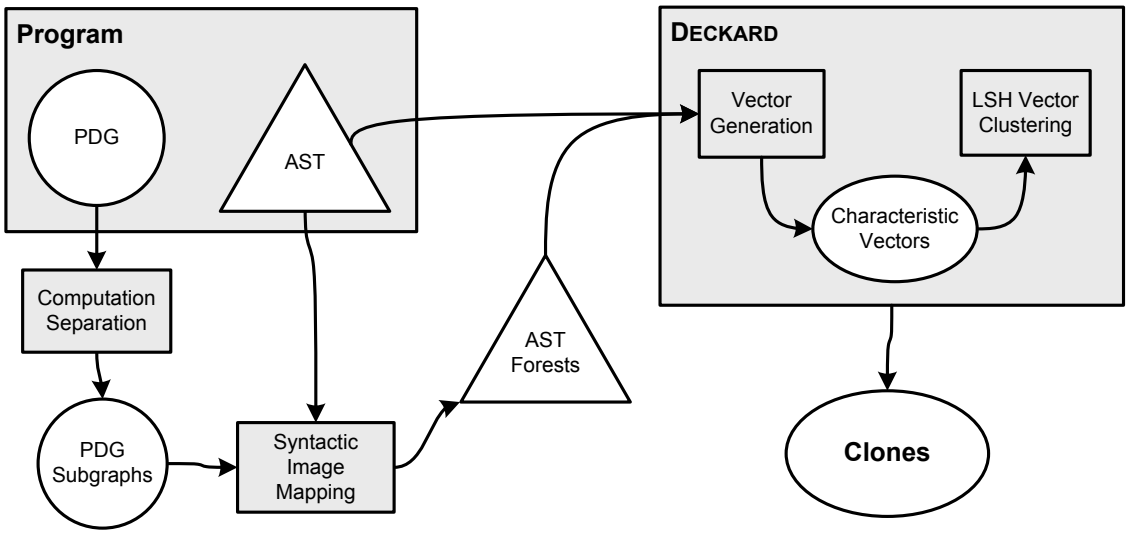


Figure 2.11: PDG-based semantic clone detection algorithm built on DECKARD.

4. We use DECKARD's sliding window vector merging to generate a complete set of characteristic vectors for each AST forest.

5. We use Locality Sensitive Hashing to quickly solve the near-neighbor problem and enumerate the clone groups. As in DECKARD, we apply a set of post-processing filters to remove spurious clone groups and clone group members.

2.4.3 Evaluation

The effectiveness of the extended algorithm is evaluated against DECKARD on five open source projects: The GIMP, GTK+, MySQL, PostgreSQL, and the Linux kernel, where PDGs are constructed using the CodeSurfer tool from GrammaTech [75].

Performance

The extended algorithm is still comparably scalable *w.r.t.* DECKARD. Table 2.4 shows the execution times for both our semantic clone detection algorithm and tree-based algorithm (DECKARD). In this table, the VGen phase performs all vector generation; Cluster phase performs LSH clustering.

Program	Size (MLoC)	AST Only (min:sec)		AST+PDG (min:sec)	
		VGen	Cluster	VGen	Cluster
GIMP	0.78	0:37	1:11	0:44	1:45
GTK	0.88	0:31	0:57	0:34	0:53
MySQL	1.13	0:27	1:16	0:29	0:34
PostgreSQL	0.74	0:40	1:50	0:51	2:30
Linux	7.31	8:42	6:01	9:48	7:24

Table 2.4: Clone detection times.

Coverage wise, our tool locates more clones than its tree-only predecessor. This is expected: we produce exactly the same set of vectors, then augment it with vectors for semantic clones. In many cases, we observed that the average number of cloned lines of code per clone group differs significantly between the tree-only and semantic versions of the analysis. As we increase the minimum number of statement nodes for a given clone group, the clones reported by the semantic analysis tend to cover more lines of code than those reported by the tree-only analysis. We believe this is because when the minimum vector size is set to smaller values, the larger semantic clones are detected simultaneously with their smaller, contiguous constituent components. While the semantics-based analysis is able to tie these disparate components together, it does not necessarily increase the coverage. When the minimum is raised, these smaller components are no longer detected as clones.

After each of our experiments, we sampled thirty clone groups at random and verified their contents as clones. When the minimum number of statement nodes was set to four, we experienced a false positive rate of two in 30. These false positives took the form of small (two to three lines) snippets of code that incidentally mapped to identical characteristic vectors. When the minimum was set to eight or more, we found no false positives in these random samples. This low false positive rate is possibly due to the relatively large magnitude of AST-based vectors: the Linux kernel code contained (after macro expansion) an average of thirty AST nodes per line. These larger vectors create a more unique signature for each line of code that is less likely to incidentally match a non-identical line.

```
1  static void os_event_free_internal(os_event_t event)
2  {
3     ut_a(event);
4     /* This is to avoid freeing the mutex twice */
5     os_fast_mutex_free(&(event->os_mutex));
6     ut_a(0 == pthread_cond_destroy(&(event->cond_var)));
7     /* Remove from the list of events */
8     UT_LIST_REMOVE(os_event_list, os_event_list, event);
9     os_event_count--;
10    ut_free(event);
11 }
```

```
1  void os_event_free(os_event_t event)
2  {
3     ut_a(event);
4     os_fast_mutex_free(&(event->os_mutex));
5     ut_a(0 == pthread_cond_destroy(&(event->cond_var)));
6     /* Remove from the list of events */
7     os_mutex_enter(os_sync_mutex);
8     UT_LIST_REMOVE(os_event_list, os_event_list, event);
9     os_event_count--;
10    os_mutex_exit(os_sync_mutex);
11    ut_free(event);
12 }
```

Figure 2.12: Example of semantic clones differing only by locking code (MySQL).

Qualitative Analysis

The quantitative results above show that this technique finds more clones with a larger average size. In addition, we took a closer, qualitative look at the results since semantic clones should be more interesting than simple copied and pasted or otherwise structurally identical code. We have observed programming idioms that are pervasive throughout the results, which shows the value of extending DECKARD's algorithm to PDG-based clones.

On a general level, our algorithm was able to locate semantic clones that were slightly to somewhat larger than their syntactic equivalents, which were also found. The semantic clone often contained the syntactic clone coupled with a limited number of declarations, initializations, or return statements that were otherwise separated from the syntactic clone by unrelated statements. In addition, many semantic clones were subsumed by larger

```
1  pg_index = heap_open(IndexRelationId, RowExclusiveLock);
2  indexTuple = SearchSysCacheCopy(INDEXRELID,
3        ObjectIdGetDatum(indexRelationId), 0, 0, 0);
4  if (!HeapTupleIsValid(indexTuple))
5    elog(ERROR, "cache lookup failed for index %u",
6         indexRelationId);

8  indexForm = (Form_pg_index) GETSTRUCT(indexTuple);
9  Assert(indexForm->indexrelid = indexRelationId);
10 Assert(!indexForm->indisvalid);
11 indexForm->indisvalid = true;

13 simple_heap_update(pg_index, &indexTuple->t_self, indexTuple);
14 CatalogUpdateIndexes(pg_index, indexTuple);
15 heap_close(pg_index, RowExclusiveLock);
```

```
1  rel = heap_open(TypeRelationId, RowExclusiveLock);
2  tup = SearchSysCacheCopy(TYPEOID,
3      ObjectIdGetDatum(typeOid), 0, 0, 0);
4  if (!HeapTupleIsValid(tup))
5    elog(ERROR, "cache lookup failed for type %u",
6         typeOid);

8  typTup = (Form_pg_type) GETSTRUCT(tup);
9  typTup->typowner = newOwnerId;

11 simple_heap_update(rel, &tup->t_self, tup);
12 CatalogUpdateIndexes(rel, tup);
13 /* Update owner dependency reference */
14 changeDependencyOnOwner(TypeRelationId, typeOid, newOwnerId);
15 heap_close(rel, RowExclusiveLock); /* Clean up */
```

Figure 2.13: Example of semantic clones differing only by debugging and unrelated code (PostgreSQL).

syntactic clones. We observed cases where our tool was able to locate clone groups that differed only in their use of global locking (*e.g.*, Figure 2.12).

We also noticed clones that differed only by debugging, logging or unrelated (data-wise) statements. One example appears in Figure 2.13. While we found several examples of this behavior, we do suspect that we missed other cases due to the fact that logging code often displays current state information, which places a data dependency on the logging code and causes its inclusion in a larger semantically related PDG subgraph.

2.5 Discussion and Future Work

This section discusses possible directions to extend our technique.

Support for q-Levels

In our current implementation of DECKARD, we mainly consider 1-level binary branches because we would like to detect reordering, such as `stmt1; stmt2;` and `stmt2; stmt1;`, as clones because 1-level does not take order information into account. As a result, some code, such as `var->bar.foo` and `var.bar->foo`, may be considered as clones while they are usually not. One way to address this issue is to allow general q-level vectors such as 2-level. However, because the number of node kinds in a grammar is commonly larger than one hundred, then using 2-level vectors would produce vectors of a very high dimension (in the order of 100^3) which may impose additional challenges on the LSH-based vector clustering in order to find close vectors efficiently. Whether such q-level vectors are appropriate may depend on applications.

In fact, the design and implementation of our vector generation algorithm has already taken some of these concerns into account. For example, a quick solution is to re-engineer the `IndexOf` function in Algorithm 2.1 to be able to generate "1.5-level" indices for certain pairs of node kinds in order to incorporate some neighboring information into characteristic vectors while remaining the vector dimension within LSH's capability. Thus, `var->bar.foo` and `var.bar->foo` will have different vectors and will no longer be clustered as clones, reducing false clone reports. In addition, under most circumstances, characteristic vectors for parse trees are sparse, and LSH has been engineered to handle sparse vectors of dimension more than 100^3 [50]. As a result, we may still be able to carry out clone detection using higher-level branch vectors.

```
1   /* we have: */
2   stmt 1;
3   stmt a;
4   stmt b;
5   stmt c;
6   stmt 2;

8   ...

9   /* and: */
10  stmt 1;
11  stmt 2;

13  /* and: */
14  stmt a;
15  stmt b;
16  stmt c;
```

Figure 2.14: An example illustrating an alternative kind of code similarity. Lines 3–5 are copied from lines 14–16. In this case, we would like to view that code between lines 2–6 is only *one* edit away from the code between lines 10–11.

Alternative Metric for Code Clones

For clone detection, we may also take a different view of similarity between code. Currently, we use tree editing distance as the target. Sometimes, it makes sense to copy a big chunk of code and insert it somewhere. For example, consider the code fragment in Figure 2.14. In the figure, we assume that lines between 14–16 are copied into lines 3–5. Because this is just a simple copy, it is reasonable to consider this as a single edit. Thus, we would like to view the code between lines 2–6 and the one between lines 10–11 as similar; more precisely, they have distance *one*.

On the other hand, were the code between lines 14–16 not there, the code between lines 2–6 should be viewed to be far apart from and lines 10–11 because many edits (insertions of additional statements) are now needed to make them the same.

To detect similarity with respect to such a view of code similarity, we can modify our existing algorithm to iteratively compute clones until we reach a fixpoint. More precisely, we first apply the existing algorithm to discover simple clones (*level-1 clones*). Then we use this information to compute *level-2 clones*, by replacing the level-1 clones with a fresh *single node* in the code and applying our existing algorithm on the modified code. This process can be repeated if necessary, but we believe in practice level-2 clones are what we really need. We leave it as future work to experiment with such enhancement of our algorithm.

Beyond Trees

As a concrete example shown in Section 2.4, our algorithm is not limited to trees; it can be adapted for code clones defined on program dependency graphs.

There are also a few other directions that our algorithm can be extended. First, we can extend it to generate implicit programming rules, similar to PR-Miner [120]. The extension should be straightforward because of the generality of our algorithm: only the vector generation routine may need to be adjusted to account for higher-level information. Similarly, we can extend our algorithm to check for correlated structure updates, which are those updates that must happen at the same time. It is also valuable to extend our algorithm to detect high-level code clones and similar subgraphs, for example, within control flow graphs or software models (*e.g.*, UML diagrams). Further, our algorithm can be applicable for binary code [154] and even articles in natural languages if we could construct meaningful, structure-preserving characteristic vectors for them using a natural language parser, *e.g.*, the Stanford Parser [2, 105, 106].

Chapter 3

Context-Based Detection of Clone-Related Bugs

Chapter 1 has mentioned that the substantial existence of code clones may incur significant maintenance costs. One aspect of software maintenance is to track bugs and their fixes across the software life cycle. Multiple copies of the same piece of code may require duplicated tracking efforts for the same bug since a bug fix for one copy of the code has better to be propagated to all other copies, and forgetting to propagate bug fixes may incur confused, duplicated debugging efforts. Also, code cloning often comes with minor modifications to the copied code in order to fit the code into a different surrounding context. Inconsistent changes may introduce subtle bugs into the copied versions. Although much research has proposed techniques for detecting and refactoring clones to improve software maintainability, little has been done to detect latent clone-related bugs.

This chapter introduces a general notion of *context-based inconsistencies* among clones and presents an efficient algorithm to detect such inconsistencies for locating clone-related bugs. We have implemented our algorithm and evaluated it on large open source projects including the latest versions of the Linux kernel and Eclipse. Many previously unknown bugs and programming style issues in both projects have been discovered. We have also

categorized the bugs and style issues and noticed that they exhibit diverse characteristics and are difficult to be detected by any single existing bug detection technique, and thus believe that the context-based inconsistencies are an important complement to previous bug detection techniques and can help reduce software maintenance costs on code clones.

3.1 Overview

Software projects contain much similar code (*i.e.*, code clones), which may be introduced by many commonly adopted software development practices, *e.g.*, reusing a generic framework, following a specific programming pattern, and directly copying and pasting code. These practices can improve the productivity of software development by quickly replicating similar functionalities. However, such practices, especially copying and pasting, can also reduce program maintainability and introduce subtle programming errors. For example, when enhancements or bug fixes are done on a piece of duplicated code, it is often necessary to make similar modifications to the other instances of the code. As previous work [115] indicates, it is easy for developers to miss some instances of the duplicated code and thus to introduce subtle bugs. "I think I have fixed the bug. Why is it still happening?" and "Why does the function work well in that way, but not in this way?" may be example questions that software maintainers ask and which may allude to clone-related bugs.

Finding similar code automatically is an important step to alleviate the aforementioned issues. Much work [19, 92, 101, 119] has been done on clone detection. Also, many techniques [90, 148] have been proposed for eliminating similar code to help reduce software maintenance cost. On the other hand, various studies [102,104,148] indicate that similarity in software is inherent, and clone unification and removal may not always be desired. There are multiple reasons for the inherent existence of clones.

Limited expressiveness of programming languages: Clone instances may have evolved over a substantial period of time with many independent changes so that they cannot be easily unified or removed.

Performance concerns: Unified code may have worse performance.

Software development practices: Some experimental code may be short-lived, unstable, and inappropriate to be unified.

Thus, code clones would always exist, and clone-related bugs may also lurk around in mature code. Therefore, we need automatic techniques, beyond clone detection and removal, to detect and eliminate such errors.

In particular, this chapter introduces a general notion of *context-based inconsistencies* among clones (to capture the intuition that similar code should be used "consistently") and develop an efficient algorithm for detecting such inconsistencies (to discover latent clone-related bugs). Our approach is based on the central observation that many bugs are caused by copying and pasting code and making minor modifications to the pasted code and its surrounding code (*i.e.*, the "context"). If the changes are not consistent with the context of the duplicated code, or if the code is pasted without appropriate changes for use in the new context, *inconsistencies* occur and may strongly indicate bugs in the code.

3.1.1 Sample Inconsistencies

Figure 3.1, 3.2, and 3.3 show several inconsistencies detected by our approach among similar code. In Figure 3.1, lines 408–419 and lines 323–334 are detected as similar code. However, the enclosing **`if`** statements of the two pieces of code are different: one uses `strncmp` which takes three arguments, while the other uses `strcmp` which takes only two arguments in their respective conditions for **`if`**. This turns out to be a logic error in "Code 2."

In Figure 3.2, lines 4861–4864 and lines 2386–2391 are detected as similar code. Their main difference is that two more statements (lines 2389–2390) in "Code 4" are enclosed in the **`for`** statement. One can see that `msgbuf[0]` in "Code 4" is always `NULL` (*i.e.*, `'\0'`), and thus nothing in `msgbuf` would be output. Although the difference does not significantly impact the functionality of the code, it is still a bug and would manifest in debugging code.

As another example, the pair of code snippets in Figure 3.3, which differ in variable

Code 1
File: linux-2.6.19/drivers/scsi/arm/eesox.c

```
407: if (length >= 9 && strncmp(buffer, "EESOXSCSI", 9) == 0) {
408:     buffer += 9;
409:     length -= 9;
410:
411:     if (length >= 5 && strncmp(buffer, "term=", 5) == 0) {
             ......
418:     } else
419:         ret = -EINVAL;
420: } else
421:     ret = -EINVAL;
```

Code 2 (Similar to Code 1 but buggy)
File: linux-2.6.19/drivers/scsi/arm/cumana_2.c

```
322: if (length >= 11 && strcmp(buffer, "CUMANASCSI2") == 0) {
323:     buffer += 11;
324:     length -= 11;
325:
326:     if (length >= 5 && strncmp(buffer, "term=", 5) == 0) {
             ......
333:     } else
334:         ret = -EINVAL;
335: } else
336:     ret = -EINVAL;
```

Figure 3.1: Sample No. 1 context-based inconsistency among similar code.

naming, exhibits a local inconsistency within the clones themselves. In particular, the **if** condition performs a `NULL` check on `l_stride`, but `r_stride` is used within the **if** statement in "Code 6." This is suspicious when one wonders what is the use of the null-check. and indeed, it has been confirmed by the GCC developers as a bug and fixed quickly.

Although such bugs may be discovered by thorough testing, designing "enough" test cases is often difficult and time consuming. In addition, even if a program exhibits abnormal behavior, it may still require much time to locate the actual bug locations. Such bugs may

Code 3
File: linux-2.6.19/drivers/cdrom/sbpcd.c

```
4859: if (cmd_type==READ_M2)
4860: {
4861:  for (xa_count=0;xa_count<CD_XA_HEAD;xa_count++)
4862:    sprintf(&msgbuf[xa_count*3], " %02X", ...);
4863:  msgbuf[xa_count*3]=0;
4864:  msg(DBG_XA1,"xa head:%s\n", msgbuf);
4865: }
```

Code 4 (Similar to Code 3 but buggy)
File: linux-2.6.19/drivers/cdrom/sbpcd.c

```
2386: for (i=0;i<response_count;i++)
2387: {
2388:    sprintf(&msgbuf[i*3], " %02X", ...);
2389:    msgbuf[i*3]=0;
2390:    msg(DBG_SQ1,"cc_ReadSubQ:%s\n", msgbuf);
2391: }
```

Figure 3.2: Sample No. 2 context-based inconsistency among similar code.

Code 5
File: gcc-4.0.1/gcc/fortran/dependency.c

```
414: if (l_stride != NULL)
415:    mpz_cdiv_q (X1, X1, l_stride->value.integer);
```

Code 6 (Similar to Code 5 but buggy)
File: gcc-4.0.1/gcc/fortran/dependency.c

```
422: if (l_stride != NULL)
423:    mpz_cdiv_q (X2, X2, r_stride->value.integer);
```

Figure 3.3: Sample No. 3 context-based inconsistency among similar code.

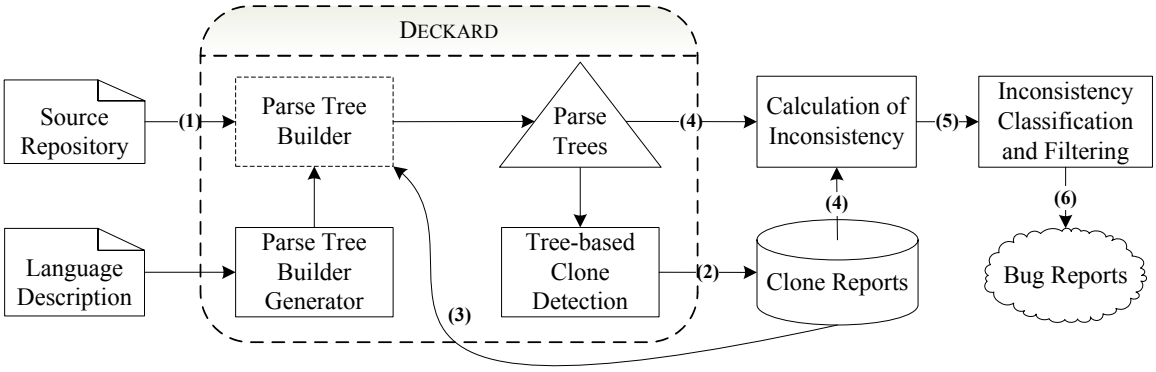


Figure 3.4: Overview of inconsistency-based bug detection approach.

also be difficult to detect using standard program analysis techniques due to a couple of reasons:

- These techniques usually require certain property specifications (*e.g.*, null-pointers cannot be dereferenced, array accesses must be within bounds, and certain temporal safety properties should hold), while clone-related bugs are diverse and difficult to specify (*cf.* Section 3.4).
- Most of these techniques still have limited scalability, especially for code bases with millions of lines of code (*e.g.*, the Linux kernel and Eclipse).

3.1.2 Approach Overview

Figure 3.4 shows the architecture and main steps of our bug detection algorithm. First, it uses a clone detection tool (DECKARD in this study) to detect code clones in programs (Steps 1 and 2). Then, it computes inconsistencies in the contexts of clones based on parse trees (Steps 3 and 4). Next, it classifies the inconsistencies based on their potential relations with actual bugs and filters out uninteresting inconsistencies (Step 5). Finally, it generates bug reports to be inspected by developers (Step 6). We describe these steps in detail in Section 3.2, and present our implementation and empirical evaluation of the approach in Section 3.4.

3.2 Algorithm Description

This section describes the details of our approach for detecting inconsistencies and bugs based on the contexts of code clones. We first define clone *contexts* based on the definitions for clones from Chapter 2 (Section 3.2.1). We then define three types of *context-based inconsistencies* among clones and formulate the inconsistency detection as mechanical tree matching problems (Section 3.2.2). We also classify these inconsistencies based on their potential relations with actual bugs (Section 3.2.3) and present heuristics for pruning uninteresting ones (Section 3.2.4). Finally, the remaining clones with un-filtered inconsistencies are considered as bug indicators and can be reported to developers for inspection.

3.2.1 Basic Definitions

For the definitions below, we assume there is a generic clone detection algorithm A such that $A(F_1, F_2) = true$ if and only if code fragments F_1 and F_2 are similar code *w.r.t.* a suitable definition of similarity (*e.g.*, in terms of tree editing distance as presented in Chapter 2).

Definition 3.1 (Clones)**.** A pair of code fragments F_1 and F_2 is called a *clone pair* if they are similar, *i.e.*, $A(F_1, F_2)$ holds. A group of code fragments $\{F_1, \ldots, F_k\}$ is called a *clone group* if $A(F_i, F_j)$ holds for all $1 \leq i, j \leq k$. Each code fragment F_i in a clone pair or a clone group is called a *clone instance*.

Definition 3.2 (Context)**.** The *context* of a code fragment F is the innermost language construct that encloses F. We further restrict contexts to control-flow constructs. For example, in C, **`if`**, **`switch`**, **`for`**, **`do`**, **`while`** statements, and function definitions are such constructs. In Java, class definitions can also be such constructs.

We use the contexts for clones as the basis for inconsistency and bug detection to capture our intuition that similar code should perform similar functionalities and should be used under similar contexts. Thus clones with different contexts indicate likely bugs. Admittedly, the actual code surrounding clones may vary, and not all differences in the

surrounding code are equally indicative of bugs. Thus, in this dissertation, we confine the context to be the smallest enclosing construct that may impact the control flows of a clone to ignore context differences that may be too far away from the clone.

We here use an example to illustrate our definitions: in Figure 3.1, the lines 408–419 and 323–334 are a clone pair, and the two **`if`** statements beginning with lines 407 and 322 are the respective contexts for the two clones.

Although the definition of a context is language-dependent, it is still straightforward to provide a generic algorithm to find the context of a given clone. Algorithm 3.1 gives the high-level description of how we find the context of a clone. Given the parse tree of the program file which contains the clone, we perform a bottom-up search in the tree to find the smallest enclosing tree node of the clone that is a contextual node, *i.e.*, a control-flow construct, and the subtree rooted at this node contains the whole piece of the given clone.

3.2.2 Context-Based Inconsistencies

We observe that bugs are often introduced when a developer duplicates a piece of code and makes inappropriate changes or forgets to make certain necessary changes. We thus formalize context differences of clones as indications of such bugs. In particular, we define three types of context inconsistencies of clones.

Definition 3.3 (Type-1 Inconsistency)**.** Given a pair of clones F_1 and F_2 and their corresponding contexts C_1 and C_2, F_1 and F_2 have a *type-1 inconsistency* if the kinds of C_1 and C_2 (denoted by $\textsc{kind}(C_1)$ and $\textsc{kind}(C_2)$), in terms of language constructs, are different. We denote such an inconsistency by $I_1(F_1, F_2)$ such that $I_1(F_1, F_2) = 1$ if $\textsc{kind}(C_1) \neq \textsc{kind}(C_2)$, and $I_1(F_1, F_2) = 0$ otherwise.

We lift this definition to a clone group. Given a clone group G, there exists a unique equivalence partition $G/I_1 = \{g_1, g_2, \ldots, g_k\}$ of G such that

(1) $\forall i\ \forall C_{\in g_i} \forall C'_{\in g_i} : I_1(C, C') = 0$, and

(2) $\forall i\ \forall j_{\neq i}\ \forall C_{\in g_i} \forall C''_{\in g_j} : I_1(C, C'') = 1$.

Code 1 (missing necessary checks in the shaded part)
File: org.eclipse.debug.ui/ui/org/eclipse/debug/ui/memory/AbstractTableRendering.java

```
3557: int colCnt = fTableViewer.getTable().getColumnCount();
3558: TableItem item = fTableViewer.getTable().getItem(0) ;
3559: for (int i=0; i<colCnt; i++)
3560: {
3561:  Point start = new Point(item.getBounds(i).x, ......
3562:  start = fTableViewer.getTable().toDisplay(start);
       ......
3565:  if (start.x < point.x && end.x > point.x)
3566:    return i;
3567: }
```

Code 2
File: org.eclipse.debug.ui/ui/org/eclipse/debug/internal/ui/memory/provisional/ \\
AbstractAsyncTableRendering.java

```
2697: TableItem item = null;
2698: for (int i=0; i<fTableViewer.getTable().getItemCount(); i++)
        item = ......
2705: if (item != null)
2706: {
2707:  for (int i=0; i<colCnt; i++)
2708:  {
2709:    Point start = new Point(item.getBounds(i).x, ......
2710:    start = fTableViewer.getTable().toDisplay(start);
         ......
2713:    if (start.x < point.x && end.x > point.x)
2714:      return i;
2715:  }
2716: }
```

Figure 3.5: Sample type-1 inconsistency and bug.

We say that G has *type-1 inconsistency* if $k > 1$ and let $I_1(G) = k$ denote the type-1 inconsistency of G.

As an example, Figure 3.5 shows a clone pair which has type-1 inconsistency. The

Algorithm 3.1 Find the context of a given clone

```
1: function CONTEXT(T : tree, F : clone): node
2:     Find the smallest subtree T_F in T, s.t., T_F properly contains F
3:     Let R be the root of T_F
4:     Find the youngest contextual ancestor node C_R of R
5:     Return C_R
6: end function
```

lines 3559–3567 and 2707–2715 are reported as a clone pair, and the context for "Code 1" is a function definition, while the context for "Code 2" is an **`if`** statement. In fact, the inconsistency was confirmed as a bug on line 3558: the developers omitted the necessary checks to make sure that `0` is a valid subscript and `item` is not `null`.

Calculating type-1 inconsistencies is as straightforward as finding contexts: we simply compare the kinds of the nodes returned by Algorithm 3.1. Despite their simplicity, type-1 inconsistencies have interesting potentials for finding many bugs, especially bugs due to missing checks (*cf.* Table 3.4).

Definition 3.4 (Type-2 Inconsistency)**.** Given a pair of clones F_1 and F_2 and the conditional predicates P_1 and P_2 in their contexts, F_1 and F_2 have a *type-2 inconsistency* if P_1 does not match P_2 in terms of parse tree matching. We denote such an inconsistency by $I_2(F_1, F_2)$ such that $I_2(F_1, F_2) = 1$ if P_1 and P_2 do not match, and $I_2(F_1, F_2) = 0$ otherwise. If F_1 or F_2 has no corresponding predicates, we let $I_2(F_1, F_2) = 0$.

We lift this definition to a clone group. Given a clone group G, there exists a unique partition $G/I_2 = \{g_0, g_1, g_2, \ldots, g_k\}$ of G such that

(1) g_0 contains exactly those clones with no context predicates,

(2) $\forall i_{\neq 0} \forall C_{\in g_i} \forall C'_{\in g_i} : I_2(C, C') = 0$, and

(3) $\forall i_{\neq 0} \forall j_{\neq 0} \neq i \; \forall C_{\in g_i} \forall C''_{\in g_j} : I_2(C, C'') = 1$.

We say that G has *type-2 inconsistency* if $k > 1$ and let $I_2(G) = k$ denote the type-2 inconsistency of G.

This definition is also language-dependent because different languages may have different definitions of conditional predicates. As an example, the pair of code snippets in Figure 3.1

has no type-1 inconsistency because both of them are **`if`** statements. However, they have a type-2 inconsistency because their **`if`** conditions invoke two different functions with different numbers of parameters.

Clones with type-2 inconsistencies may be executed along different control flow paths (which are controlled by the conditions) and thus behave differently. Such inconsistencies violate our assumption that similar code should perform similarly under similar situations, and thus may indicate bugs.

A simple way to compare two conditional predicates P_1 and P_2 is to compare every node in the parse trees for P_1 and P_2 in a pre-order traversal. We say P_1 matches P_2 if each node in one parse tree has the same type (without considering certain terminal values, *e.g.*, identfier names and literal constants) as its correponsding node in another tree. Although such a comparison may falsely report inconsistencies on semantically equivalent but syntactically different expressions, such as `p[i]` and `*(p+i)` in C, it may provide a reasonable upper-bound estimation on the total number of type-2 inconsistencies in clone groups. One can also argue that as long as the purpose of duplicating code is to improve software productivity (instead of plagiarism), code clones with the same semantics should often have the same syntactic structure and there is usually no reason to modify the code to have different syntactic structures. In practice, some code may become similar due to other reasons besides direct copying and pasting (*e.g.*, applying the same programming pattern). Thus semantically equivalent but syntactically different expressions do exist, and in Section 3.2.4, we employ certain heuristics to reduce false alarms on type-2 inconsistencies.

Definition 3.5 (Type-3 Inconsistency)**.** Given a pair of clones F_1 and F_2, F_1 and F_2 have a *type-3 inconsistency* if F_1 and F_2 contain different numbers of *unique* identifiers. We denote such an inconsistency by $I_3(F_1, F_2)$ such that $I_3(F_1, F_2) = 1$ if F_1 and F_2 have different numbers of unique identifiers, and $I_3(F_1, F_2) = 0$ otherwise.

We lift this definition to a clone group. Given a clone group G, there exists a unique equivalence partition $G/I_3 = \{g_1, g_2, \ldots, g_k\}$ of G such that

(1) $\forall i\ \forall C_{\in g_i} \forall C'_{\in g_i} : I_3(C, C') = 0$, and

(2) $\forall i\ \forall j_{\neq i}\ \forall C_{\in g_i} \forall C''_{\in g_j} : I_3(C, C'') = 1$.

We say that G has *type-3 inconsistency* if $k > 1$ and let $I_3(G) = k$ denote the type-3 inconsistency of G.

The type-3 inconsistencies capture another kind of differences in code clones that may be introduced by modifying identifiers (including names of variables, functions, types, etc.), which is a common practice during copying and pasting code. Often, not all identifiers in clones are modified; occasionally some identifiers that should be changed are left unchanged, and some that should not be changed are changed. These cases may lead to different numbers of unique identifiers in the clones and thus indicate likely bugs. For example, in the pair of code in Figure 3.3, "Code 6" is similar to "Code 5," but it has seven unique identifiers (excluding keywords and punctuations), while "Code 5" only has six. In fact, it was confirmed by the GCC developers that the "extra" identifier (`r_stride`) should have been `l_stride` instead.

Compared with type-1 and type-2 inconsistencies, type-3 inconsistencies are local to code clones themselves. We calculate the type-3 inconsistencies by traversing the parse trees of clones and counting all identifiers that we visit. Alternatively, a simpler lexical scanner can be used to count the numbers. We currently do not distinguish identifiers for types from identifiers for variables or functions. Based on the parse trees, we can incorporate such differences to improve the accuracy of type-3 inconsistencies.

3.2.3 Classification of Inconsistencies

It is obvious that not all context inconsistencies are actual bugs. In fact, probably most of such inconsistencies are not bugs when code is copied and pasted with caution. To better invest manual efforts when examining the inconsistencies for bugs, we utilize a series of classification heuristics to rank the inconsistencies so that we can examine most likely buggy inconsistencies first, or filter out unlikely buggy clones to reduce false positives.

The first criterion for sorting inconsistent clone groups is the types of their inconsistencies.

Definition 3.6 (Inconsistency Rank)**.** Given a clone group G, the *inconsistency rank* of the group, denoted by $\textsc{rank}(G)$, is a 4-tuple $\langle |G|, I_1(G), I_2(G), I_3(G)\rangle$, where $|G|$ is the number of clones in G.

Given two clone groups G_1 and G_2 and their associated ranks:

$$\textsc{rank}(G_i) = \langle |G_i|, I_1(G_i), I_2(G_i), I_3(G_i)\rangle, \qquad i \in \{1, 2\},$$

the order between G_1 and G_2 is given by the lexicographical order between $\textsc{rank}(G_1)$ and $\textsc{rank}(G_2)$, *i.e.*:

$$\begin{cases} G_1 = G_2 & \iff & |G_1| = |G_2| \bigwedge \forall i \in \{1,2,3\}\ I_i(G_1) = I_i(G_2) \\ G_1 > G_2 & \iff & \begin{cases} |G_1| > |G_2| & \bigvee \\ |G_1| = |G_2| \bigwedge I_1(G_1) > I_1(G_2) & \bigvee \\ |G_1| = |G_2| \bigwedge I_1(G_1) = I_1(G_2) & \\ \qquad\qquad \bigwedge I_2(G_1) > I_2(G_2) & \bigvee \\ |G_1| = |G_2| \bigwedge \forall i \in \{1,2\}\ I_i(G_1) = I_i(G_2) & \\ \qquad\qquad \bigwedge I_3(G_1) > I_3(G_2) & \end{cases} \end{cases}$$

Recall that for $i \in \{1, 2, 3\}$, $I_i(G) > 1$ indicates the existence of type-i inconsistencies in the clone group G. The larger $I_i(G)$ is, the more inconsistencies the group has. However, an $I_i(G)$ that is too high (*e.g.*, > 5) and too close to the total number of clones in the group (*e.g.*, $> 50\%$ of $|G|$) may mean that there are too many inconsistencies in the clone group. In such cases, the inconsistencies may be intended by developers, and may no longer be indications of anomalies or bugs. On the other hand, the smaller $I_i(G)$ is (except for one), the more likely the inconsistencies are not intended and are indications of bugs. Based on such an intuition, we choose to include only those clone groups G with small values of $I_i(G)$ during the ordering of clone groups.

In addition, based on our experience, type-1 inconsistencies may be further classified into several subtypes, and different subtypes have different likelihoods to be bugs. In the following, we define subtypes for type-1 inconsistencies. Such a type-refinement can further help the classification of clone groups and reduce false positives (Section 3.2.4).

Definition 3.7 (Inconsistency Subtypes)**.** Given a clone pair F_1 and F_2 and their contexts C_1 and C_2, the kinds of C_1 and C_2 (in terms of language constructs) can be one of *switch*, *if*, *loop*, *function-definition* (or *fundef*), and *program* (or *prog*). The subtype of F_1 and F_2, written $I_S(F_1, F_2)$, is defined based on the kinds of C_1 and C_2:

$$
\begin{array}{lll}
\textbf{Subtype-1}: & I_S(F_1,F_2) = 1, & \text{if } \textsc{kind}(C_1) = (\mathit{fundef} \mid \mathit{prog}) \bigwedge \\
 & & \textsc{kind}(C_2) = (\mathit{fundef} \mid \mathit{prog}) \\
\textbf{Subtype-2}: & I_S(F_1,F_2) = 2, & \text{if } \textsc{kind}(C_1) = (\mathit{fundef} \mid \mathit{prog}) \bigwedge \\
 & & \textsc{kind}(C_2) = \mathit{loop} \\
\textbf{Subtype-3}: & I_S(F_1,F_2) = 4, & \text{if } \textsc{kind}(C_1) = \mathit{loop} \bigwedge \\
 & & \textsc{kind}(C_2) = (\mathit{switch} \mid \mathit{if}) \\
\textbf{Subtype-4}: & I_S(F_1,F_2) = 8, & \text{if } \textsc{kind}(C_1) = \textsc{kind}(C_2) = \mathit{loop} \\
\textbf{Subtype-5}: & I_S(F_1,F_2) = 16, & \text{if } \textsc{kind}(C_1) = (\mathit{switch} \mid \mathit{if}) \bigwedge \\
 & & \textsc{kind}(C_2) = (\mathit{switch} \mid \mathit{if}) \\
\textbf{Subtype-6}: & I_S(F_1,F_2) = 32, & \text{if } \textsc{kind}(C_1) = (\mathit{switch} \mid \mathit{if}) \bigwedge \\
 & & \textsc{kind}(C_2) = (\mathit{fundef} \mid \mathit{prog})
\end{array}
$$

Given a clone group G, the subtype of G is the bit-wise OR of all possible subtype inconsistencies among the clones in G, *i.e.*,

$$I_S(G) = \mathsf{OR}_{F_i,F_j \in G} \; I_S(F_i, F_j).$$

The subtypes capture our intuitions on the relations between context inconsistencies and latent bugs:

- Subtype-6 may indicate a missing conditional check or a redundant check.

- Subtype-5 and subtype-4 are actually type-1 consistent, but their conditional predicates within different contexts may help refine possible type-2 inconsistencies (*i.e.*, different conditional predicates).

- Subtype-3 and subtype-2 may indicate that a substantial semantic change is intended among the clones and the code may be less likely a bug.

- Subtype-1 may indicate that the clones and their contexts have too few differences to introduce a bug.

Also, one can utilize more language-dependent features to refine the above subtypes. For example, the kinds of contexts in Java may also include `synchronized` and `try-catch-finally`. If a clone in a clone pair misses such a context, it may indicate lock-based concurrency errors or un-handled exceptions.

The inconsistency ranks and subtypes form the basis of the following filtering heuristics for bug detection.

3.2.4 Filtering Heuristics

Many reasons, such as different programming styles, may introduce context inconsistencies that may not be actual bugs. For example, preferences to **`while`** loops over **`for`** loops may introduce context differences; device driver code for different models of a printer may be similar but have different conditional checks for different features of the printers. For bug detection, such inconsistencies are usually false positives and should be pruned before manual inspection. We next present a set of heuristics based on inconsistency ranks and subtypes to prune clone groups that are unlikely bugs.

The first heuristic is to prune certain type-1 inconsistencies by considering some contexts as the same:

`for` $\equiv$ **`while`**: we treat **`for`** and **`while`** as the same context.

`switch-case` ≡ `if-else`: we treat a **`switch-case`** statement and a sequence of **`if-else`** statements as the same context.

`fundef` ≡ `classdef` ≡ `file`: we treat function definitions, class definitions, and file scopes as the same context.

The second heuristic is to prune type-2 inconsistencies by recognizing some small expressions that are likely semantically equivalent:

`ce` ≡ `ce!=0`: we treat a conditional expression `ce` the same as the expression `ce!=0` (using C's syntax).

`!ce` ≡ `ce==0`: we treat a conditional expression `!ce` the same as the expression `ce==0` (using C's syntax).

`e1<e2` ≡ `e2>e1`: we treat conditional expressions of the form `e1<e2` the same as `e2>e1`, where `e1`, `e2` are two expressions.

`e1+e2` ≡ `e2+e1`: we treat `e1+e2` the same as `e2+e1` because addition is conceptually commutative; similar treatment is applied to other conceptually commutative operators, such as `*`, `||`, and `&&`).[1]

`.` ≡ `->`: we treat different field access operators, such as `.` and `->`, the same, and ignore address-of and dereference operators, such as `&` and `*`.

In addition, we also propose several filtering heuristics to prune clone groups. These heuristics are based on the observation that some types of inconsistencies do not strongly indicate bugs because of either too minor changes or too significant changes among the clones. Given a clone group G, we have the following filters:

[1] In reality, `e1<e2` and `e1+e2` may not be semantically equivalent to `e2>e1` and `e2+e1`, respectively, because the C language specification leaves the order of evaluation of subexpressions in an expression undefined. Also, `e1&&e2` and `e1||e2` may not be semantically equivalent to `e2&&e1` and `e2||e1`, respectively, because the evaluation of logical expressions is short-circuited.

Filter 1: If subtype-1 is set in $I_S(G)$, prune the group since such cases may imply that the clones have no real differences.

Filter 2: If subtype-2 is set in $I_S(G)$, prune the group since such cases may imply that the clones are intended to have significant semantic differences because adding or removing loops is unlikely accidental.

Filter 3: If subtype-3 is set in $I_S(G)$ and the *if* or *switch* context is not enclosed in another *loop* context, prune the group since such cases may imply that the clones may be intended to be semantically different because of loops.

Filter 4: Instead of using the exact tree matching algorithm (Section 3.2.2) to compute type-2 inconsistencies, use more approximate measures, such as tree editing distances or Euclidean distances [92], to allow small differences in contextual conditions to further prune type-2 inconsistencies.

Filter 5: If G has type-3 inconsistencies and the difference among the numbers of unique variables in the clones in G is large (*e.g.*, > 2), prune the group since such cases may imply that the clones have gone through many modifications and possibly have different semantics.

Filter 6: If the clones in G are very close to each other (*e.g.*, less than 10 lines apart), prune the group since such cases may imply that the clones were written by the same programmer during a short period of time and thus may be less likely to contain inconsistencies.

After filtering, the remaining clone groups can be inspected for actual bugs. We will show that the estimated amount of inspected code is small *w.r.t.* the sizes of the original programs. For example, we manually examined less than 12000 lines of code in the Linux kernel, which are collectively about 0.2% of the total 5.6 million lines, to find 57 bugs and programming style issues (*cf.* Section 3.4). Considering that the maintenance of duplicated

code is still mostly manual and little work has been done on finding clone-related bugs, the code inspection burden of our approach is light and worthwhile, especially when compared to manual audits of the entire code base.

We note that it is possible that our filters may prune certain buggy inconsistencies. This is a common trade-off one needs to make: less code inspection burden versus finding more bugs. Section 3.4 will present results to show that the filters perform well in terms of reducing false positives with few false negatives.

3.2.5 Complexity Analysis

Our approach relies on detected clones and traverses the parse trees of the clones to calculate inconsistency ranks and subtypes. In the worst case, our approach takes linear time *w.r.t.* the number of clone groups, the size of a clone group, and the sizes of the parse trees of the source files that contain the clones. In practice, most clone groups are small, with two to five clone instances; few groups have more than ten clone instances. In addition, our algorithm often only needs to traverse a small portion of the parse trees because clones are usually much smaller than the complete files. Our experimental results in Section 3.4 will also show that our approach can often generate valuable bug reports in tens of minutes, including clone detection time.

3.3 Implementation

Our bug detection algorithm works on top of a clone detection tool. In our implementation, we use our own DECKARD (Chapter 2) to detect code clones as inputs to our bug detection algorithm. DECKARD's tree-based and language-independent nature make it a convenient choice for our purpose. However, it is worth mentioning that our inconsistency-based bug detection algorithm is general and can be applied with other clone detection techniques [9, 10, 18, 19, 101, 110, 114, 119]. Although those techniques have algorithmic and parametric differences from DECKARD, we do not anticipate any fundamental difficulty in using them

with our algorithm.

On the other hand, quality and quantity of detected clones clearly affect the effectiveness of our approach. As a summary for Chapter 2, DECKARD has three main parameters that may affect the number and quality of its detected clones. The first one is *Similarity*, which specifies the similarity between two pieces of code for them to be considered clones. It ranges from 0.0 to 1.0; the larger *Similarity* is, the less difference is tolerated among clones, and less clones may be reported. The second parameter is minT, the minimum token number for a piece of code to be considered. The larger minT is, the less clones may be reported. The third parameter, stride, mainly controls the minimum spatial distance (in terms of tokens in source files) between two clones. The smaller its value is, the more clones may be reported. Smaller strides may also produce more overlapping clones, and the post-processing phase in DECKARD may take more time to prune overlapping segments. If stride is set to ∞, only non-overlapping and syntactically complete pieces of code (*e.g.*, a complete **`if`** statement or a complete **`for`** statement) are considered as candidates for clones.

3.4 Empirical Evaluation

Our experiments are mainly performed on a machine with a 3GHz Intel Xeon CPU, 8GB of memory, and Fedora Core 5. Section 3.4.1 describes the setup for our empirical evaluation and Section 3.4.2 presents detailed results on detected clone-related inconsistencies and bugs.

3.4.1 Experimental Setup

First, for most of our evaluation, we set DECKARD's *Similarity* parameter to 1.0, minT (the minimum token number) to 50, and stride to ∞. These correspond to standard choices in other clone detection tools, and we want to focus on evaluating the bug detection aspects of our approach. We note that the numbers of false positives and negatives may vary with

Program	Version	# Files	# LoC	# Clone Groups	# Cloned LoC	Detection Time (sec)
Linux	2.6.19	8733	5639833	7852	358331	289
Eclipse	CVS 01/08/07	8320	1832332	2246	70455	160

Table 3.1: Statistics of subject programs and their clones used in inconsistency studies.

different parameter settings. In Section 3.5, we will evaluate the impact of different choices of DECKARD's parameters on the effectiveness of our approach for bug detection.

Second, we choose well-known large open source projects, such as the Linux kernel and Eclipse, as the subjects in our evaluation. These projects are written in different programming languages, C and Java, which can help us evaluate the generality and language-independence of our approach. Table 3.1 shows some basic statistics on the projects, including their lines of code and numbers of source files. Table 3.1 also shows clone-related metrics. For each project, it lists the number of clone groups detected by DECKARD, the total number of lines of cloned code, and DECKARD's time on clone detection. Thus, the 358331 lines of clones in the 7852 clone groups in the Linux kernel and the 70455 lines of clones in the 2246 clone groups in Eclipse form the main code base where we search for bugs in our following experiments.

3.4.2 Results of Inconsistency and Bug Detection

This section presents statistics of the inconsistencies and bugs detected by our approach in the Linux kernel and Eclipse. It also provides a categorization of the bugs we found and compares our results with another bug detection tool–CP-Miner.

Statistics of Detected Inconsistencies

Our approach found many context inconsistencies in our subject programs. Many of these inconsistencies revealed true programming errors and programming style issues.

Table 3.2 shows how many inconsistencies and bugs we found in the subject programs. For each clone group reported by DECKARD, its inconsistency rank and subtype were cal-

Program	Detection Time (sec)	# Clone Groups	# Type-1		# Type-2		# Type-3	
			Inc.	Bugs	Inc.	Bugs	Inc.	Bugs
Linux w/ all filters	387	7852	115	10	350	25	69	12
Linux w/o filters	355	7852	177	11	527	29	388	15
Eclipse w/ all filters	127	2246	146	2	249	13	26	2
Eclipse w/o filters	125	2246	224	4	390	17	91	4

Program	Total #		# Suspects	# Style Issues	# False Positives	Estimat. of LoC for Inspection
	Inc.	Bugs				
Linux w/ all filters	396	33	69	9	285	11258
Linux w/o filters	881	41	85	16	739	37430
Eclipse w/ all filters	265	15	42	13	195	6096
Eclipse w/o filters	461	21	50	17	373	11536

Table 3.2: Numbers of inconsistencies and bugs reported when all or no bug filters (Section 3.2.4) were enabled.

culated (*cf.* Section 3.2.2), and we counted the number of clone groups of each type of inconsistencies (Columns "# Type-i Inc.") and the total number of groups reported as potential bugs (Column "Total # Inc."). We use the number of lines of code (Column "Estimat. of LoC for Inspection") in all the groups, including their contexts, to estimate the amount of code that we need to inspect for actual bugs. Such numbers may help readers to understand better the amount of manual effort to inspect the inconsistent clone groups. The amount of code ranges from 0.2% to 0.7% of the original programs, or from 3.2% to 16.2% of the clones. We believe the manual effort can be justified by the large number of detected bugs.

The numbers of actual bugs revealed by each type of inconsistencies are shown in Columns "# Type-i Bugs." The total numbers of bugs, programming style issues, and suspicious clones are also shown in Columns "Total # Bugs," "# Style Issues," and "# Suspects" respectively. For each remaining clone group after filtering, we manually inspected it to check whether it points to a real bug. We made such decisions based on our knowledge of the code:

- If we have high confidence that an inconsistency causes inappropriate behavior in any

clone of the group, we classified it as a bug.

- If we have high confidence that an inconsistency has no effect on the intended behavior of the clones, we classified it as a false positive.
- If we believe the clones are behaviorally correct but the code has redundancies or is unnecessarily complicated or confusing, we classified it as a programming style issue.
- If we are uncertain about an inconsistency or it takes us too long (more than 30 minutes) to understand the code, we classified it as a suspect.

During the examination of a clone group, we may also perform simple data-flow analysis to help understand the code. For most clone groups, the code was fairly easy to understand and the manual inspection took only several minutes each.

We were able to find 33 bugs and 9 programming style issues in the Linux kernel and 15 bugs and 13 style issues in Eclipse when all filters were enabled. When fewer filters were enabled, we were able to find more bugs and style issues (Row "Linux w/o filters" and "Eclipse w/o filters" in Table 3.2). Table 3.3 also shows the impact of different filters (*cf.* Section 3.2.4) on bug detection for the Linux kernel. With no filter enabled or all filters enabled, more than 450 false positives were pruned with 15 false negatives. This is a trade-off one has to make between low false positive and negative rates. The bugs exhibit diverse characteristics (Table 3.4), and they would be difficult for existing bug detection tools to discover. Considering the relatively light code inspection that is needed, we believe our approach is worthwhile for improving quality of the programs. To date we have received confirmation from developers for two bugs in the Linux kernel and two bugs in Eclipse (and additional ones for GCC and Apache) for the bugs that we have reported.

Table 3.2 also shows the running time of our algorithm (Column "Detection Time (sec)"), excluding the time for clone detection and manual inspection. Most of the time was spent on (re-)parsing of clones, the most expensive operation in our approach. As a possible implementation improvement, we could keep parse trees in memory when they

Filter	# Inconsistencies				#	# Sus-	# Style	Est. of	# False
	Type-1	Type-2	Type-3	Total	Bugs	pects	Issues	LoC	Positives
All.	**115**	**350**	**69**	**396**	**33**	**69**	**9**	**11258**	**285**
1	177	527	98	591	40	83	13	16495	455
2	133	485	383	837	39	82	16	36396	700
3	159	506	388	859	40	84	16	37061	719
4	177	445	388	807	38	80	14	35214	675
5	176	524	356	849	41	85	16	34151	707
6	165	474	324	767	38	80	13	34265	636
None.	**177**	**527**	**388**	**881**	**41**	**85**	**16**	**37430**	**739**

Table 3.3: Effects of filters on false positives and negatives. Each row corresponds to different filters (Section 3.2.4). "None" means no filter was enabled; "All" means all filters were enabled. They are the same data for Table 3.2.

were generated by DECKARD for the first time to reduce the number of re-parsing, trading space for time.

Breakdown of Bugs and Style Issues

In this section, we categorize the detected bugs and programming style issues in the Linux kernel and Eclipse (Table 3.4 and 3.5). In total, there are 41 bugs and 16 style issues in the Linux kernel, and 21 bugs and 17 style issues in Eclipse. We also noticed that the bugs and style issues have diverse characteristics, confirming that many different kinds of bugs can be introduced when developers copy and paste code.

Table 3.4 lists the main reasons that caused these bugs. Missing necessary conditional checks before using certain data seems to be the most common kind of bugs (Row "ID 1"). Figure 3.5 shows such an example. Figure 3.6 shows another error caused by "Wrong function calls." Lines 2674–2721 and lines 2724–2773 are clones, and they have different numbers of unique identifiers. It did not take us long to realize that the call to `pci_bus_write_config_word` on line 2682 should have been `pci_bus_write_config_byte`. Because `pci_bus_write_config_word` takes parameters of type `void *`, the type checker did not catch the mismatch between the type of `temp_byte` and the expected type by the function. At a coarser granularity, most bugs caused by "Wrong function calls," "Wrong variables," "Wrong data fields," and "Wrong macros" may be classified as "Wrong identi-

ID	Category	# Bugs in Linux	# Bugs in Eclipse
0	Total	41	21
1	Missed conditional checks	9	8
2	Negated conditions	1	0
3	Inappropriate conditions	1	3
4	Off-by-one	2	1
5	Inappropriate scoping	2	0
6	Missed or inappropriate qualifiers	2	0
7	Wrong variables	3	4
8	Missed or inappropriate locks	4	0
9	Inappropriate logic for corner cases	3	2
10	Unhandled cases or exceptions	2	3
11	Wrong function calls	3	0
12	Wrong data fields	5	0
13	Wrong macros	4	0

Table 3.4: Categories of detected clone-related bugs.

fiers." The fact that many bugs fall into this category confirms that copying and pasting code often requires identifier renaming, which can be error-prone.

Table 3.5 shows the kinds of style issues found by our approach. Although some code with style issues may be deliberate, such as for debugging, for code obfuscation, for an experimental or immature feature, or as dummy code, we believe that code with the style issues listed in Table 3.5 is generally confusing, results in less optimized code, and reduces program readability and maintainability, and it should be avoided as much as possible.

Here, we give an example for "Less optimized code" in Figure 3.7. "Code 1" and "Code 2" were reported as clones, but have different context conditions. One can see that `newWidth` in "Code 1" is calculated more times than necessary (line 592), while "Code 2" is optimized to calculate `width` only once (line 680). Compilers may not be able to perform the optimization automatically because `getClientArea()` is fairly complicated and the compiler may not be able to infer that `newWidth` is a constant.

Some of the bugs and style issues can be detected by existing techniques. For example, missing a `NULL` check (*e.g.*, Figure 3.5) can be revealed by data flow analyses. However,

Code 1 (wrong function call)
File: linux-2.6.19/drivers/pci/hotplug/cpqphp_ctrl.c

```
2673: if (hold_IO_node && temp_resources.io_head) {
          ......
2681:     temp_byte = (hold_IO_node->base) >> 8;
2682:     rc = pci_bus_write_config_word(..., temp_byte);
            ......
2700:       temp_byte = (io_node->base - 1) >> 8;
2701:       rc = pci_bus_write_config_byte(..., temp_byte);
        ......
2721: }
```

Code 2
File: linux-2.6.19/drivers/pci/hotplug/cpqphp_ctrl.c

```
2724: if (hold_mem_node && temp_resources.mem_head) {
          ......
2732:     temp_word = (hold_mem_node->base) >> 16;
2733:     rc = pci_bus_write_config_word(..., temp_word);
            ......
2751:       temp_word = (mem_node->base - 1) >> 16;
2752:       rc = pci_bus_write_config_word(..., temp_word);
      ......
2773: }
```

Figure 3.6: Clone-related bug example: a wrong function call.

many bugs may involve programming logic errors, such as inappropriate conditions (*e.g.*, Code 2 in Figure 3.1) and inappropriate scoping (*e.g.*, Code 4 in Figure 3.2), and are difficult to discover without specifications. Section 3.5 discusses further how our approach and existing techniques may complement each other.

We also believe that the categories of clone-related bugs and style issues can be useful in two aspects: they can help developers to understand better possible reasons that cause clone-related errors and consciously prevent them from happening again in the future; and automated tools may be implemented to check code clones against each of such categories

ID	Category	# Style Issues	
		in Linux	in Eclipse
0	Total	16	17
1	Redundant conditional checks	1	5
2	Redundant locks	2	0
3	Dead code	0	0
4	Unnecessary obscured code	2	1
5	Less optimized code	2	2
6	Redundant macro checking code	1	0
7	Unhandled application features	2	5
8	Unused variables	3	0
9	Redundant operations	1	0
10	Redundant type casts	1	1
11	Unnecessary name/data aliases	1	1
12	Inconsistencies between code and comments	0	1
13	Redundant error checking code	0	1

Table 3.5: Categories of detected style issues in code clones.

for code validation.

Breakdown of False Positives

Admittedly, our approach reported many false positives although it found many actual bugs. False positive rates, in terms of the number of bugs and style issues over the number of identified inconsistencies, may be up to 90%. On the other hand, many bugs discovered by our approach may be difficult to find with other techniques, and the number of lines of code of the reported inconsistencies account for only less than 1% of the total number of lines of code in the original programs. We believe the manual effort involved in applying our approach is worthwhile for improving program reliability. Next, we analyze possible reasons for the false positives so that we can reduce them further in the future.

Table 3.6 lists several reasons that are responsible for most false positives in our experiments. Basically, many differences among clones legitimately exist because they are intended to behave differently, such as drivers for devices with slightly different features, and exception handling code for different types of exceptions. Any such intended behavioral

Code 1 (less optimized)
File: eclipse-cvs/org.eclipse.swt/Eclipse SWT/gtk/org/ \\
eclipse/swt/widgets/ExpandBar.java

```
590: for (int i = 0; i < itemCount; i++) {
591:   ExpandItem item = items [i];
592:   int newWidth = Math.max (0,getClientArea().width – spacing*2);
593:   if (item.width != newWidth) {
594:     item.setBounds (0, 0,newWidth, item.height, false, true);
595:   }
596: }
```

Code 2
File: eclipse-cvs/org.eclipse.swt/Eclipse SWT/gtk/org/ \\
eclipse/swt/widgets/ExpandBar.java

```
680: int width = Math.max (0, getClientArea().width – spacing*2);
681: for (int i = 0; i < itemCount; i++) {
682:   ExpandItem item = items [i];
683:   if (item.width != width)
         item.setBounds(0, 0, width, item.height, false, true);
684: }
```

Figure 3.7: An example of programming style issues: less optimized code.

differences may cause a false positive because our approach is currently syntax-based:

- Our current definitions for contexts and inconsistencies only consider language syntax.
- All our filters are mainly syntax-based.
- The clone detection tool used in our approach, DECKARD, is also syntax-based and may report semantically different but syntactically similar code as clones.

All of the reasons for false positives listed in Table 3.6 are related to program semantics (*e.g.*, types, data and control dependencies) and their intended behavior. It would be interesting to extend the idea of context-based inconsistency and bug detection to semantic-based

#	Reasons for False Positives
1	Different features in devices cause divergences in their (mostly similar) driver code.
2	Similar functions accept parameters of different types and handle types differently.
3	Names of types, functions, variables, *etc.* clash.
4	Some code of similar and simple syntactic structures may not be real clones.

Table 3.6: Category of inconsistencies that cause false positives.

clones and incorporate semantic information into the definitions of contexts and inconsistencies and the filters to detect bugs more accurately (discussed further in Section 3.5)

Comparison with CP-Miner

CP-Miner [119] is a token-based clone detection tool for C. To our knowledge, it is the only existing tool that looks for bugs directly in cloned code. In this section, we compare CP-Miner's effectiveness with DECKARD's on the Linux kernel. Section 5.2.4 will discuss other related bug detection techniques.

CP-Miner also assumes that inconsistencies among clones indicate bugs. However, its definition of inconsistencies is *local* to the clones, similar to our type-3 inconsistencies. Different from our type-3 inconsistencies, it is based on *identifier mappings* among clones: Given a clone pair F_1 and F_2, every instance of all identifiers in F_1 is mapped to an identifier in the same position in F_2; and for each unique identifier `ID`, an *UnchangedRatio*(`ID`) is defined as the following:

$$\textit{UnchangedRatio}(\texttt{ID}) \triangleq \frac{\#\ \textit{of Unchanged}(\texttt{ID})\ \text{in}\ F_2}{\textit{Total}\ \#\ \textit{of}(\texttt{ID})\ \text{in}\ F_1}$$

For example, in the clone pair in Figure 3.3, let Code 1 be F_1 and Code 2 be F_2, then *UnchangedRatio*(`X1`) $= \frac{0}{2} = 0$ because both instances of `X1` have been changed to `X2`. Similarly, we have *UnchangedRatio*(`l_stride`) $= \frac{1}{2}$ and *UnchangedRatio*(`value`) $= 1$. Also, the order of C_1 and C_2 matters. Similar to our type-3 inconsistencies, *UnchangedRatio* is used to measure whether programmers change identifiers consistently when they copy and paste code. A non-zero or non-one value for *UnchangedRatio* may indicate inconsistent

changes of the identifiers and reveal a potential bug. *UnchangedRatio* is a finer-grained metric than our type-3 inconsistencies, and if a clone pair has type-3 inconsistency, it must have some identifier with a non-zero value for its *UnchangedRatio*, which means CP-Miner may generate more reports than ours and we may miss certain bugs. On the other hand, our type-3 inconsistencies are more efficient to calculate and report fewer false positives.

Table 3.7 shows our experiments on the Linux kernel 2.6.19, using 50 for minT (the minimum token number) and 1.0 for *Similarity* for both CP-Miner and DECKARD. We also set the stride parameter in DECKARD to ∞. DECKARD reported fewer clones (Column "# Cloned LoC") in slightly longer time (Column "Total Run Time (sec)"), thus the initial code base for reporting bugs is smaller for our approach.[2] However, our approach still found more bugs and style issues (Column "# True Pos.")[3] because we look for inconsistencies not only within clones, but also in the contexts which are beyond the clones. We also achieved a much lower false positive rate. All reports (Column "# Positives") generated by CP-Miner and our approach were manually inspected by us. Among the 251 reports from CP-Miner, 55 cases were classified as suspects.

It is also interesting to note that the intersection between the problems found by CP-Miner and the problems found by our approach is empty (Column "Set Diff. of True Pos."). Among the 13 cases from CP-Miner, five (three were duplicated reports) were pruned by our filters, and the other eight were not in the clones reported by DECKARD with our parameter setting. After examining these eight reports, we see no reason why they could not have been detected by our type-3 inconsistencies if they had been reported as clones by DECKARD with different parameter settings. Thus, it would also be interesting to investigate further whether the finer-grained identifier mapping-based approach in CP-Miner can actually detect more bugs than our simpler type-3 inconsistencies.

[2]These results do not imply DECKARD performs worse than CP-Miner in general. With different parameter settings, DECKARD can detect more clones than CP-Miner in the same amount of time (*cf.* Section 2.3.3).

[3]CP-Miner does not report cases when *UnchangedRatio* > 0.4 by default. It is also a trade-off between false positives and negatives chosen by CP-Miner.

	Total Run Time (sec)	# Cloned LoC	# Positives	# Suspects	# True Pos. (Bug+Style)	Set Diff. of True Pos.
Our Approach	676	358331	396	69	42	42
CP-Miner	582	534202	251	55	13	13

Table 3.7: Comparison with CP-Miner on Linux kernel 2.6.19.

3.5 Discussion

We now discuss issues related to our approach's effectiveness.

Which Clones to Choose From?

Our approach works on code clones detected by DECKARD (Chapter 2) which is a tree-based clone detection tool. The set of clones may vary when we use different parameters for DECKARD, and the bug reports from our approach may also vary. Table 3.8 shows such effects by varying DECKARD's parameters.

Recall from Section 3.3 that the three main parameters for DECKARD are *Similarity*, minT (the minimum token number), and stride. We experimented with different similarities (the first segment of Table 3.8, by setting the minimum token number to 50 and the stride to ∞), different minimum token numbers (the second segment of Table 3.8, by setting *Similarity* to 1.0 and stride to ∞), and different strides (the third segment of Table 3.8, by setting *Similarity* to 1.0 and minT to 50) on the Linux kernel 2.6.19.

As a summary, smaller similarities, smaller minimum token numbers, and smaller strides will lead to more clones, and our approach will also produce more bug reports. It would be interesting to actually calculate the false positive and negative rates for each of the parameters and give a more quantitative guide on choosing appropriate parameters for different applications. According to our experience, *Similarity* 1.0, minT 50, and stride ∞ had a good balance between false positives and negatives.

	Time (sec)		# Clone	# LoC	Est. of LoC	# Inconsistencies			
	Clone	Detection	Groups	(clones)	for Inspc	Type-1	Type-2	Type-3	Total
Similarity (minT 50, stride ∞)									
1.0	**289**	**387**	**7852**	**358331**	**11258**	**115**	**350**	**69**	**396**
0.999	288	360	7854	367272	9481	110	280	64	330
0.99	290	402	8462	403545	13945	122	322	155	441
0.95	311	837	15738	599866	63684	788	1919	2089	2637
minT (*Similarity* 1.0, stride ∞)									
50	**289**	**387**	**7852**	**358331**	**11258**	**115**	**350**	**69**	**396**
128	277	108	1324	161079	840	2	13	6	18
64	294	370	7805	294931	6393	55	147	83	220
32	330	1042	23780	495037	23990	389	1040	427	1327
16	372	3602	63763	867991	182957	3552	8311	6514	11485
stride (*Similarity* 1.0, minT 50)									
∞	**289**	**387**	**7852**	**358331**	**11258**	**115**	**350**	**69**	**396**
16	345	507	10828	433778	15044	159	390	154	499
8	370	694	15536	520857	22649	218	500	286	705
4	419	1184	26532	675863	34235	439	782	569	1174
2	517	2199	49235	916752	50862	723	1399	1136	2199

Table 3.8: Potential effects of different clone detection parameters on false positives and negatives with all filters enabled.

Why Not Use Existing Bug Detection Techniques?

Many static and dynamic analysis techniques, such as ESC/Java [63] and Valgrind [134], exist for bug detection. Static analyses are usually sound—they do not miss bugs with the property that they are looking for. Dynamic analyses are usually accurate—they do not report false positives. However, such techniques usually need to analyze all code in a program for bugs because they do not know in general *where* to analyze, and thus may not be able to scale to programs with millions of lines of code. Also, they usually require certain property specifications so that they can know *what* kinds of bugs to target, and thus their bug finding capabilities are limited by available specifications.

Compared with those techniques, our approach has mainly two advantages: it effectively reduces the amount of code which requires analysis for bugs; and it can hint at possible properties of latent bugs for more specific analyses through the discovered inconsistencies. For example, when the type-3 inconsistency in Figure 3.3 was discovered, a simple difference

analysis of the data and control dependencies of the two snippets revealed that there is a missing data dependency between `r_stride` and the **`if`** condition. Then we knew that the latent bug could either be a missing `NULL` check on `r_stride` or a wrong use of `r_stride`. Such advantages can help guide the existing techniques on *where* and *what* to analyze and make them more scalable. In fact, many bugs we found are difficult to be discovered by any single existing technique. We believe that our approach complements well the existing techniques. Conversely, incorporating existing analysis techniques into our approach can provide semantic information to help reduce more false positives and improve the usability of our approach. The following section elaborates on this.

How to Reduce False Positives Further?

Currently, when a clone group is reported as a possible bug, we inspect it in the following steps: first locate the clones in the original source code and find the actual differences among clones based on their inconsistency ranks and subtypes, then inspect the clones and their contexts to look for any hints, *e.g.*, comments and data dependencies, which can explain the differences, and perform manual data-flow analysis to help understand the code whenever necessary.

Several steps in the inspection process can be automated and may help to prune false positives without human intervention. On one hand, we frequently asked ourselves common questions, such as "where is the variable defined," "whether can the return value of this function ever be null," and "whether can this conditional predicate ever be false," during code inspection. Most of such semantic-related questions can be easily answered by many program analyses and theorem proving techniques, and help to decide whether an inconsistency is legitimate. As for the purpose of filtering, such techniques do not need to be accurate as long as they can answer the questions with low false negative rates. On the other hand, the inconsistencies among clones can provide hints at what questions to ask. As a simple example, there is a missing `NULL` check for variable `item` in "Code 1" in Figure 3.5. Based on the difference, it was obvious to ask whether `item` could ever be `NULL`

to decide whether the missed check is an actual bug. Generalizing such question-generation schemes and integrating them with other techniques will be like integrating query generators with answer machines, and it will be interesting to investigate how many more false positives may be pruned by an automated code inspection mechanism provided by such an integration.

As another aspect, our current definitions of contexts and inconsistencies are mainly *syntax*-based and only consider the smallest enclosing control-flow construct of a clone. They have not incorporated any semantics of the clones, and neither do the filters for pruning bug reports. It will be interesting to extend our definitions to *semantic*-based representations of programs, such as program dependency graphs [62], so that semantic information, such as types, data and control dependencies, can be considered to help detect more bugs while pruning more intended inconsistencies. Further, we believe that the basic idea that inconsistencies among clones are indications of bugs can be directly applied to semantic-based code clones [110, 114], which are most robust against code modifications, such as re-ordered statements, non-contiguous code, and redundant code, than syntax-based clones. Such clones, together with syntax-based clones, may naturally exclude syntactically similar but semantically different code and thus introduce fewer false positives in the first place.

What Conditions Decide Applicability?

A basic assumption that we have made in this dissertation is that similar code should perform similar functionalities under similar contexts and thus context inconsistencies among code clones can be strong indications of bugs. However, in practice, much similar code does not satisfy such an assumption. Many inconsistencies among clones are likely intended and should not be treated as indications of bugs. If such inconsistencies commonly occur in a program, our approach would report too many false positives to be useful.

One such situation is when we use smaller similarities to generate clones. When a smaller similarity is used, code with more differences may still be treated as clones, and

thus inconsistencies can become more commonly intended. In our experiments, we mainly restricted DECKARD's *Similarity* to 1.0 to avoid clones with too many differences. Although such a restriction may miss certain bugs, we believe that it currently is a reasonable trade-off between low false positive and negative rates. In the future, our inconsistency classification and filtering heuristics can be improved to tolerate inconsistencies which are introduced by smaller similarities so that false positive rates can be kept low.

Another situation is when clones evolve independently and intentionally deviate from each other in certain aspects. For example, drivers for several different models of a display card from the same manufacturer have much code in common, but also have many differences that handle different features in the different models. Such inconsistencies among clones may only be indications of different features instead of bugs. For such cases, simple filtering strategies may not always be enough for reducing false positives because the inconsistencies caused by diverse code features may not be easily described by any specific filtering pattern. If there exists *inconsistency specifications* from people who know what kinds of differences are intended, we could utilize such specifications to find unintended inconsistencies only and reduce false positives. With the advances in specification mining techniques [5, 113], we may be able to infer such inconsistency specifications in the near future, instead of asking for them from developers.

Chapter 4

Scalable Mining of Functionally Equivalent Code Fragments

The studies presented in previous chapters focus on similar code based on code syntactic structures and dependency graphs. Although such similar code is an important class of duplicated code and the main target of previous studies, there is another important class of duplicated code that is concerned with the actual functionality of the code and beyond syntax trees and dependency graphs. Such duplicated code can occur due to various software engineering practices, *e.g.*, n-version programming. Although there have been studies on coarse-grained, program-level and function-level functional equivalence at small scales, it is not known whether significant fine-grained, code-level functional duplications exist. Detecting functional equivalence is desirable also because it could enable many applications including code understanding, optimization, maintenance, and reuse.

This chapter introduces the first practical approach to automatically mine functionally equivalent code fragments of arbitrary size—down to an executable statement—from large programs. The notion of functional equivalence in this chapter is based on the input and output behavior of each piece of code. The core algorithm is developed based on automated random testing. It automatically extracts a large number of candidate code fragments from

a subject program, and generates random inputs to partition the code fragments based on their output values on the generated inputs. A large-scale empirical evaluation of the algorithm is conducted on the Linux kernel 2.6.24. The results show that there may exist many *functionally equivalent* code fragments that are *syntactically different* (*i.e.*, they are unlikely due to code copying and pasting practices).

4.1 Overview

It is a common intuition that similar code, either syntactically or semantically, is ubiquitous in large software projects. Large-scale studies have shown that a large project may often contain more than 20% syntactically similar code. The abundance of similar code provides opportunities for studies on its origins, characteristics, and evolution with the potential to improve many aspects of software development processes.

Although there are many existing techniques for detecting *syntactically similar* code [9, 19, 92, 101], few studies exist that target *semantically similar* code that may not be syntactically similar. In fact, no study has even empirically validated the ubiquitous existence of semantically similar code although it is a common intuition.

This chapter proposes a scalable approach for identifying *functionally equivalent* code fragments, where functional equivalence is a particular case of semantic equivalence that is concerned with the input and output behavior of a piece of code. With such an approach, we are able to discover many functionally equivalent code fragments, covering more than 624K lines of code in the Linux kernel 2.6.24, confirming the common intuition. About 58% of the functionally equivalent code fragments are syntactically different, which shows the need for functionality-aware code analysis techniques in addition to syntactic approaches. We have also validated our results by sampling reported equivalent code fragments and running additional random tests on them. Regardless of certain limitations, more than 96% of the sampled code fragments remained in *some* equivalent clusters, showing probabilistically high accuracy of our approach.

Different from previous studies on finding semantically equivalent code or checking the semantic equivalence between two pieces of code, our approach is distinguishable in several aspects. First, the definition of functional equivalence used in this chapter considers only the equivalence among the final outputs of different code fragments given the same input; it does not consider the intermediate program states, while many definitions for semantic equivalence, *e.g.*, equivalent operational semantics, consider every intermediate program state as well. One important practical benefit of this definition is that it focuses on externally observable behavior of a piece of code and is insensitive to code transformations or different implementations for the same behavior. Thus, it may admit more functionally equivalent code.

Second, inspired by Schwartz's randomized polynomial identity testing [159], we apply random testing on arbitrary pieces of code and detect those with the same input and output behavior. The Schwartz-Zippel lemma [159, 193] states that evaluating two given polynomials with a limited number of random values are sufficient to decide, with high probability, whether the two polynomials are equivalent. Although the lemma only holds for polynomials, we leverage its intuition here for arbitrary code: if two pieces of code always produce the same outputs on a selected number of random inputs, we have high confidence that they are *functionally equivalent*; even if they may actually differ sometime, *e.g.*, at error-handling and boundary cases, they may still be considered functionally *similar* and provide opportunities for further studies.

Third, to the best of our knowledge, the large-scale study presented in this chapter is the first of its scale on the existence of functionally equivalent code in million-line software. Many unique optimizations in the implementation made our approach scalable.

The rest of the chapter is organized as the following. Section 4.2 gives an overview of our approach and discusses its algorithmic details. Then, Section 4.3 presents the implementation of our approach for C programs, and Section 4.4 details our empirical evaluation of the approach on both a small sorting benchmark and the Linux kernel. Section 4.5 discusses some limitations of and future work for our approach.

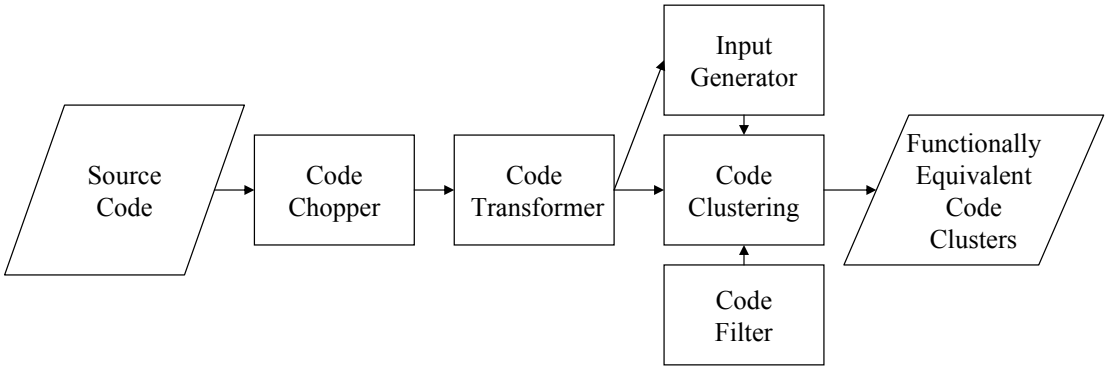


Figure 4.1: The work flow for mining functionally equivalent code.

4.2 Algorithm Description

This section presents details of our approach for detecting functional equivalence. We start with a high-level view of the approach.

4.2.1 A High-level View

The main components of our approach are illustrated in Figure 4.1.

Code Chopper Since we consider functionally equivalent code of various sizes, instead of whole programs or whole functions, we use a *code chopper* in Figure 4.1 to extract code fragments from a program as candidates for functionally equivalent ones. It takes a function definition and parses it into a sequence of statements; then it extracts all possible *consecutive subsequences* from the statement sequence, and each of the subsequences is considered a candidate for functional equivalence. We here illustrate what code fragments may be extracted for the following sample code excerpt from a selection sort algorithm: where the code to the right is the normalized sequence of statements of the code to the left.

```
1  min = i;
2
3  for(j=i+1; j<LENGTH; j++)
4  {
5
6     if(data[j] < data[min])
7        min = j;
8  }
9  if (min > i) {
10    int tmp = data[min];
11    data[min] = data[i];
12    data[i] = tmp; }
```

```
1  min = i;
2  j = i+1;
3  while (1) {
4  if(j >= LENGTH)
5  break;
6  if(data[j] < data[min])
7  min = j;
8  j++; }
9  if(min > i) {
10 tmp = data[min];
11 data[min] = data[i];
12 data[i] = tmp; }
```

When we require the minimum number of *primary statements*[1], contained in a code fragment to be 10, the code chopper will generate three code fragments if the control boundaries of statements are respected: the first contains lines 1–12, the second contains lines 2–12, and the third contains lines 3–12. If the boundaries are not respected, the code chopper may generate six code fragments.

Code Transformer Because we define semantic equivalence in terms of input and output behavior, we need to identify the inputs and outputs for each code fragment. This task is straightforward if we only consider coarser-grained code: for a whole program, we could directly use the inputs and outputs of the program; for a whole function, we could use the arguments of the functions as its inputs and its side effects and return values as its outputs. It is not obvious for a code fragment of arbitrary sizes. The *code transformer* exploits the heuristic that inputs should be the variables that are used but not defined in the code, and outputs should be the variables that are defined but not used by the code. Thus, specialized data-flow analyses can be utilized to identify such input and output variables. For example, for the code fragment containing lines 1–12 from the above example, the variables `i` and `data` are used before their first definitions and thus they are identified as the input variables for this code fragment; the variable `data` is the only variable along the control flow paths of the code fragment that is not used after its last definition and thus it is identified as the

[1] Informally, we consider every expression statement, **`if`**, **`switch`**, loop, **`return`**, **`break`**, **`goto`**, and **`continue`** statements primary. The setting for different applications may be changed.

only output variable.

Since our approach requires executions of all of the code fragments, each code fragment should be made compilable and executable. For C programs, this can involve many details such as defining all types used in the code, defining functions that are called but not defined in the code, and declaring all variables used in the code. The code transformer also takes care of these details.

Input Generator Since the executions of each code fragment require random inputs, the *input generator* component takes a code fragment and its input variables and generates random values for the input variables. Currently, it does not take the validity of randomly generated inputs *w.r.t.* a code fragment into account since we assume functionally equivalent code fragments should exhibit the same behavior on even invalid inputs. Section 4.2.5 has more details about the way our input generator works.

Code Clustering The *code clustering* component takes a set of code fragments that have been compiled and random inputs generated by the input generator, executes each code fragment with the same random inputs, and separates two code fragments into two different code clusters whenever the outputs of the two code fragments differ on the same input. In such a way, all the code fragments will be dispatched into a set of code clusters, each of which may be considered a functionally equivalent class, assuming enough executions are performed for the code fragments. Section 4.2.6 has more details on how the outputs are compared against each other and how the clustering process works.

Code Filter Since the code chopper may extract many code fragments that overlap with each other in terms of their locations in the original source code they correspond to, it may not be interesting to consider them functionally equivalent if two code fragments overlap too much. Thus, a *code filter* can be placed both before and after code clustering to reduce both unnecessary executions of code fragments and false positives.

4.2.2 Equivalence Definition

This section formally introduces our definition for the aforementioned functional equivalence. We denote a piece of code as C and its set of *input variables* as $\mathcal{I}$. We also use I to represent a *sequence* of concrete values (also called *an input*) that can be used to instantiate $\mathcal{I}$ and execute C. Similarly, we use $\mathcal{O}$ to denote the set of *output variables* of C, and use O to represent a *set* of concrete values (also called *an output*) that is a concrete instantiation of $\mathcal{O}$. Then, $C(I) = O$ means that the execution of C with the input I generates the output O. We also use $\mathbb{I}$ and $\mathbb{O}$ to represent the sets of all possible inputs and outputs respectively.

Definition 4.1 (Functional Equivalence)**.** *Two code fragments C_1 and C_2 are functionally equivalent if there exist two permutations $p_1, p_2 : \mathbb{I} \rightarrow \mathbb{I}$, such that $\forall I \in \mathbb{I}$, $C_1(p_1(I)) = C_2(p_2(I))$, where "=" is the standard set equivalence operator.*

The definition has several aspects worthy of discussion:

- Considering the fact that the ordering of input variables for two functionally equivalent code fragments should not matter, the definition allows different permutations of an input for C_1 and C_2. For example, one can see that `x1` in the following code functions the same as `y2` and `y1` functions the same as `x2`, but they appear in different orders in the headers of `foo` and `bar`. Then, for any given input $I = \langle i_1, i_2 \rangle$, we need to instantiate `<x1,y1>` as $\langle i_1, i_2 \rangle$, but `<x2,y2>` as $\langle i_2, i_1 \rangle$, in order for the two pieces of code to generate the same (in the sense of set equivalence) outputs. Adding the permutation functions in the definition is to allow different orderings of input variables among functionally equivalent code.

```
foo(int x1, int y1) {
   a1 = x1 + y1;
   b1 = x1 - y1;
}
```

```
bar(int x2, int y2) {
   a2 = y2 - x2;
   b2 = y2 + x2;
}
```

  On the other hand, considering that the ordering of input variables in any *individual* code fragment is fixed, the definition requires the *same* permutation functions for all

inputs, *i.e.*, p_1 and p_2 should be fixed for all inputs for the same pair of functionally equivalent code fragments.

- An output of C_1 and that of C_2 are compared as sets, instead of sequences. This flexibility accommodates code fragments that perform the same computation but output their results in different orders. The need is also illustrated by the above code example: given an input $I = \langle i_1, i_2 \rangle$, `foo` outputs $\langle i_1 + i_2, i_1 - i_2 \rangle$ according to the sequential ordering of `a1` and `b1` and `bar` outputs $\langle i_1 - i_2, i_1 + i_2 \rangle$, thus it is necessary to compare the outputs as unordered sets so that `foo` and `bar` can be detected as functionally equivalent.

- Considering that different code fragments can perform the same computation with different numbers or types of input variables, the definition defines the behavior of a code fragment C *w.r.t.* an input I (a sequence of concrete primitive values), instead of the input variables $\mathcal{I}$ of C. For example, the following code can be considered functionally equivalent to the above `foo` and `bar`:

```
fun( struct {int x3, y3;} X ) {
   a3 = X.x3 + X.y3;
   b3 = X.x3 - X.y3;
}
```

  Also, despite the differences in their input variables, we can use any $I = \langle i_1, i_2 \rangle$ to instantiate the only input variable `X` in `fun` as `X=`$\{i_1$`,`$i_2\}$ (in the syntax of `C` language).

  Thus, although the definition requires the inputs used for different code fragments to be the same, it does not require them to have the same numbers or types of input variables. Similarly, an output of a code fragment is viewed as a set of concrete primitive values, instead of possibly ordered or complex values for output variables of different types.

- In addition, we assume all side effects of each code fragment can be captured by its output variables and each fragment interacts with its environment only through $\mathcal{I}$

and $\mathcal{O}$.

Sections 4.2.6 and 4.3 will describe our strategies for realizing the definition for mining functionally equivalent code in practice.

4.2.3 Code Chopping

As mentioned in Section 4.2.1, code fragments are extracted from each function for later steps. Given a sequence of n primary statements, there may be $\frac{n(n+1)}{2}$ consecutive subsequences of the statements. Since code fragments that across statement boundaries, *e.g.*, the fragment containing lines 2–11 from the code snippet on Page 86, are syntactically invalid and may not be interesting units for functionality study, our code chopper thus avoids generating such subsequences. Also, we use a parameter *minStmtNum* to exclude code fragments that contain fewer than *minStmtNum* primary statements. An obvious benefit of these two options is that it helps reduce the number of candidate code fragments and relatively improves the scalability of our approach.

Algorithm 4.1 illustrates the mechanism of the code chopper. Given a syntax tree of a `C` function, it utilizes a pre-order traversal (S) of the primary statements in the function and a sliding window controlled by a starting point (s_i) and an ending point (s_j) on the statement sequence to generate code fragments that respect statement boundaries (Line 10) and *minStmtNum* (Line 13).

4.2.4 Code Transformation

The main tasks for the code transformer are to identify the input and output variables of each code fragment extracted by the code chopper and make it compilable and executable.

Input Variables

Since variables in `C` code are often required to be initialized (*i.e.*, defined) before their uses, a variable in a code fragment should get its value from the environment and thus be treated

Algorithm 4.1 Code Fragment Generation

```
1: function CodeGen(F)
2: Input: F: a function in a syntax tree
3: Output: a set of code fragments C = {c_i}
4:     C ⇐ ∅
5:     S ⇐ (pre-order traversal of all statements in F)
6:     for all statement s_i ∈ S, where i is the index of s_i in S do
7:         c_i ⇐ ⟨⟩        /* empty list */
8:         for all statement s_j ∈ S, where j ≥ i do
9:             c_i ⇐ append(c_i, s_j)
10:            if s_j, s_i not in the same statement scope then
11:                continue
12:            end if
13:            if j − i + 1 ≥ minStmtNum then
14:                C ⇐ C ∪ {c_i}
15:                vectorGen(c_i) /* Only for evaluation purpose in Section 4.4.3 */
16:            end if
17:        end for
18:    end for
19: end function
```

as an input variable if it is not defined in the code fragment before its first use. Hence, comes the following definition:

Definition 4.2 (Input Variables)**.** *A variable v used in a code fragment c is an input variable for c if there is no definition for v before* some *use of v in c, where "before" or "after" is measured along the directions of any control flow path in c.*

Liveness analysis [138] for a function F can tell us which variables should be live at the entry point of F, *i.e.*, undefined before their first uses in F and thus the input variables for F. Similarly, we use a *local* version of liveness analysis for any code fragment c extracted from F to decide which variables are live at the entry point(s) of c and should be the input variables for c. The local liveness analysis is the same as standard backwards-may liveness analysis except that it propagates liveness information only on a subgraph of the control flow graph of F that corresponds to c.

Functions called in c are also live at the entry point(s) but handled differently from variables (discussed later in this section). Undefined labels in **goto** statements imply the

target statements are not contained in the code fragment c and we can terminate the executions of c whenever they reach such **goto**s. Thus, we simply transform **goto**s with undefined labels to "`goto _dummy_label;`", and add, as the last statement of c, a labeled empty statement "`_dummy_label: ;`".

Output Variables

Given an arbitrary code fragment c, it is non-trivial to decide which variables hold the data intended by the programmer to be externally observable (*i.e.*, part of its output). We use the following heuristics to make the decision:

- A definition d for a variable v in c should serve some purpose (*i.e.*, to be used somewhere in c or later): if v is used after d, v may not be needed any more since its value has served some purpose; if v is not used after d, it should be an output variable if we want d to be used somewhere later.
- **return** statements in c may indicate that the programmer wants the return value to be part of an output. Thus, we transform all **return** statements in c so that a specially-named variable is assigned the return value before each **return** and considered as an output variable for c.

Definition 4.3 (Output Variables)**.** *A variable v in a code fragment c is an output variable for c if it is a specially-named variable for a* `return` *statement in c or there is no use of v after a definition of v on* some *control flow path in c.*

Reaching definition analysis [138] for a function F can tell us which definitions may reach the exit point of F and thus be the output for F. Similarly, we use a *local* version of reaching definition analysis for any code fragment c extracted from F to decide which variable definitions may reach the exit point(s) of c and should be the output variables for c. The local reaching definition analysis is the same as standard forwards-may reaching

definition analysis except that it propagates reaching information only on a subgraph of the control flow graph of F that corresponds to c.

We can also strength Definition 4.3 by changing *some* control flow path to *all* control flow path, then the reaching definition analysis will be a forwards-must analysis, and only those variable definitions that must reach the exit point(s) of c will be included in the set of output variables. The alternative definition will obviously change the output of a code fragment, and may affect the results of mining functionally equivalent code. Section 4.4.3 will mention an example for this effect in our benchmark program.

Type Definitions

To make a code fragment c compilable, the first thing is to define every type referenced in c. One option is to traverse the code and identify which types are used in c and search in the source files for the definitions of the used types. Since a used type may refer to another type not explicitly used in c, we need to compute a closure of the referenced types. In this dissertaion, we adopt the following simpler option which may include extra unused types: the GCC preprocessor is invoked on the original source file f from which c is generated, then the preprocessed file naturally contains all types defined in any file included by f; Thus, as long as as the code chopper includes *all* the preprocessed type definitions with c, the problem is resolved, as long as the original file is compilable.

Function Calls

Each code fragment c may call other functions, some of which are library functions, some of which are functions defined somewhere else in the original source code. Strictly speaking, for two code fragments to be functionally equivalent, we should take the side-effects of the function calls into account and include all those function definitions with c.

In this dissertaion, we take an alternative look at function calls: we view each callee as a random value generator and ignore its side-effects besides assignments through its return values (*i.e.*, the random values). Thus, n function calls in c are viewed as n extra input

variables for c whose values will be generated randomly. Such a strategy helps to limit the execution time of each code fragment and improve the scalability of our approach. As future work, it may be possible to replace function calls with a learned mapping between inputs and outputs for the callees (*i.e.*, a summary of the behavior of the callees) to model them more accurately and modularly but still keep our approach scalable.

4.2.5 Input Generation

For each execution of a code fragment c, we need to instantiate its input variables with an input that may also be used for other code fragments. To make it easier to instantiate different types of input variables, we only separate input variables into two categories (non-pointers and pointers, and arrays are treated as pointers) and our input generator aims to generate generic values that may be used for all types. We thus encode each input as a sequence of concrete values, each of which may be assigned to a variable of primitive types, and a sequence of $p0$ and $p1$, each of which may indicate a null or non-null pointer.

For example, an input $I = \{48, -1, p1, p0\}$ is able to instantiate variables of different types in the following way: if a variable v is of type **`float`**, v will be instantiated with 48.0; if v is of type **`char`**, v will be the character '0' (ASCII code 48); if v is of type

```
struct node {int value; struct node * next;}
```

v will be a struct containing `value=48` and a non-null `next` pointing to another struct, allocated at run-time, containing `value=-1` and a null `next`. If an input contains fewer values than required by a variable or a set of variables, zeros are used. For example, if v is of the following type:

```
struct three {int a; char b; float c;}
```

v will be a struct containing `a=48`, `b=EOF` (ASCII code `-1`), `c=0.0`, and the $p1$ and $p0$ are not used in this case.

With such an encoding scheme, generating random inputs and instantiating input variables can be separated into two phases. It helps the input generator to generate random inputs independently from any code fragment.

On the other hand, considering certain code specific properties may help the input generator to generate inputs more effectively and help reduce invalid code executions in the following code clustering step. In particular, we consider (1) increasing the probability of generating $p0$ against that of generating $p1$ and (2) choosing the number of generated values in an input that may suit the need of a code fragment the best.

To achieve (1), we consider limiting the probability of generating non-null pointers to avoid generating deeply linked data structure, *e.g.*, trees, and help limit code execution time. We use exponential decay on the probability of generating $p1$: the more pointer values are added in an input, the more unlikely for $p1$ to occur, *i.e.*, when generating the first pointer value for an input, the probability of generating $p1$ is $\frac{1}{2}$; when generating the second pointer value, the probability of generating $p1$ will be $\frac{1}{2^2}$; and so on.

To achieve (2), we consider generating enough concrete, primitive values for instantiating as many input variables as possible, while limiting the number of possible input permutations that may be required to check functional equivalence as defined in Definition 4.1. We thus statically estimate the possible number of concrete primitive values needed by a code fragment by counting the needed values if the "first-level" pointers are non-null (the counters are initialized to 0):

- If a variable is of a primitive type, the counter will be increased by one.
- If a variable is a struct, the counter will be increased by the number of concrete values needed by all the *non-pointer fields* in the struct. This rule is recursively applied to a non-pointer field if the field is also a struct.
- If a variable is a pointer, the counter will be increased by the number of concrete values needed by a variable of the pointed type, which is then recursively counted using these

three rules. Note that pointers pointed to by a pointer are recursively followed, but pointer fields in a struct are ignored. This is what we mean by "first-level" pointers.

The number of needed pointer values is estimated similarly:

- If a variable is of a primitive type, the counter will be kept the same.
- If a variable is a struct, the counter will be increased by the sum of the number of *pointer fields* and the number of pointer values needed by other *non-pointer fields* in the struct. This rule is recursively applied to a non-pointer field if the field is also a struct.
- If a variable is a pointer, the counter will be increased by one plus the number of pointer values needed by a variable of the pointed type, which is then recursively counted using these three rules. Similarly, pointers pointed to by a pointer are recursively counted, but pointer fields in a struct are only counted once.

Then, the number of values needed in an input for a set of code fragments is determined by the minimum among the estimations for all code fragments. Using such a minimum helps avoid redundant values in inputs and reduce the number of input permutations and code executions that may be required to check functional equivalence among all the code fragments. Section 4.3 will also introduce a code fragment grouping strategy that may help to accommodate code fragments that require inputs of different sizes.

4.2.6 Code Execution and Clustering

The goal of the code clustering component is to execute every code fragment generated and transformed in previous steps and separate them into functionally equivalent code clusters. Algorithm 4.2 illustrates the code execution and clustering process.

Algorithm 4.2 uses what we call *representative-based partitioning* strategy to make the code execution and clustering process incremental and scalable. Notice that any difference

Algorithm 4.2 Code Execution and Clustering

```
1: function CodeExe(𝕀, C)
2: Input: 𝕀: a finite set of inputs
3: Input: C: a finite set of code fragments
4: Output: a set of code clusters ℂ = {C_i}
5:     ℂ ⇐ ∅
6:     for all I ∈ 𝕀 do
7:         for all c ∈ C do
8:             𝕆 ⇐ ∅
9:             for all permutation p of I do
10:                𝕆 ⇐ 𝕆 ⋃ c(p(I)) /* code execution */
11:            end for
12:            for all C_i ∈ ℂ do /* code clustering */
13:                /* 𝕆_i is the representative outputs for C_i */
14:                if 𝕆 ⋂ 𝕆_i ≠ ∅ then
15:                    C_i ⇐ C_i ∪ c
16:                    break
17:                end if
18:            end for
19:            if ∀C_i ∈ ℂ, c ∉ C_i then
20:                C_{|ℂ|+1} ⇐ {c}
21:                𝕆_{|ℂ|+1} ⇐ 𝕆 /*record representative outputs*/
22:                ℂ ⇐ ℂ ⋃ C_{|ℂ|+1}
23:            end if
24:        end for
25:    end for
26: end function
```

between the outputs of two code fragments should cause the two fragments to be separated into two clusters, and given the output set $\mathbb{O}$ of a code fragment c on a given input I and an existing cluster C_i, we only need to compare $\mathbb{O}$ with the output set $\mathbb{O}_i$ of the representative code fragment in C_i to decide whether c can be put into C_i, avoiding quadratic number of comparisons (Lines 13–17). To make the algorithm incremental, we design it in such a way that it does not execute one code fragment on all generated inputs; instead, it tries to execute all code fragment (Line 7) on one input first and partition them into smaller sets (also called code clusters). Thus, the whole set can be gradually partitioned into functionally equivalent clusters with more and more inputs (Line 6). Also, the outputs of the representative code fragment (the first code fragment put into the cluster) are kept (Lines

19–23) for comparison with coming code fragments during the incremental partitioning.

A difficulty with the incremental scheme is to find *one* permutation of *all* inputs that satisfies the requirements of Definition 4.1. We observe that it is unlikely for two functionally different code fragments to produce the same output even when *different* permutations for different inputs are allowed. We thus relax Definition 4.1 in Algorithm 4.2 to reflect the observation and look for a permutation for *each* input independently (Lines 9–18). Section 4.3 also presents a strategy to reduce the complexity introduced by $n!$ permutations.

During output comparison, we use concrete values of output variables except for pointers for which we use $p0$ or $p1$, depending on whether the pointer is null or non-null, and the values pointed to by the pointer (recursions may occur if the pointer is multi-level).

Note that we do not consider input validity: randomly generated inputs may not satisfy implicit invariants required by each code fragment, and thus an execution of a fragment may not generate any output due to various reasons, *e.g.*, segmentation faults and infinite loops. We use error code of a failed execution as its output, and compare the error codes to decide whether two failed executions behave the same.

It is also worth mentioning that the executions of different code fragments for the same input can be easily parallelized, and so can the execution and clustering for each code cluster. Thus, the algorithm can be implemented as a parallel program and its degree of parallelism increases as it makes progress on code clustering.

4.3 Implementation

We have implemented our approach as a prototype, called EqMiner. This section discusses our implementation strategies for EqMiner.

The code chopper, the code transformer, and the input generator are implemented based on CIL [133]. The code clustering component is implemented as Python and Bash scripts.

Data Storage All the code fragments and their inputs and outputs are stored in plain text for convenient inspection. A significant disadvantage is that it may take a large amount of disk space when the number of code fragments is large even if each text file is very small. Also, since a file system often limits the number of subdirectories and files in a directory, we added a layer of file management in the code chopper and the code clustering to split or merge directories as needed. A future improvement will be to store compressed files in database system to avoid slow file operations.

Code Compilation Although it is easy and convenient to make each code fragment independent from each other, it can waste a lot of disk space and take much longer time to compile if many code fragments include common contents, such as the required type definitions. When millions of code fragments are generated, it is worthwhile to extract the common contents from the code fragments and compile the common contents just once as shared libraries, and link the share libraries with much smaller code fragments. Such an optimization was justified in our evaluation on the Linux kernel (Section 4.4): `GCC` may take about a tenth of a second on our system to compile one code fragment without any optimization; it may take up to four second if `-O3` is enable. Then, it would take at least seven days to compile millions of code fragments or nine months if `-O3` is enable. Extracting common contents from code fragments and compiling common contents only once as shared libraries, we could compile sequentially all the code fragments with `-O3` enabled in about seven days, and we utilized our cluster system with varying numbers of available hosts to finish all the compilation within 15 hours in parallel.

Input Generation The main complexity in Algorithm 4.2 is to use all possible $n!$ permutations of an input containing n values to execute each code fragment c. We argue that the exponential number of input permutations is largely unnecessary, based on the following assumptions:

- Random orderings of input variables mostly occur when computations on the variables

are (conceptually) associative, such as addition and sorting. For such associative computations, different input permutations should lead to the same output.

- Most computations in the code fragments are not associative, and when they are not, programmers are more likely to follow certain customs (such as the ordering of involved operands or the flow of computation) to order the input variables, instead of randomly.

In EqMiner, we impose an empirical limit 5! on the number of permutations allowed for each input (based on the numbers of input variables from the Linux kernel, *cf.* Section 4.4.3) Also, we no longer perform regular permutations on an input since it is better to randomly select the limited permutations from all possible permutations. For this purpose, we use *random shuffling* of an input to implement the Line 9 in Algorithm 4.2 as:

for all (upto 5!) a random shuffle p of I **do**

Also, to allow converting a randomly generated concrete value into different primitive types in C, we limit the range of the value to $[-127, 128]$ so that it can be casted into many types, *e.g.*, **`char`**, **`short int`**, **`unsigned int`**, **`float`**, etc.

Parallelization We also observe that certain properties of each code fragment, such as the numbers of input and output variables, may provide opportunities for higher degrees of parallelism. The intuition is that useful functionally equivalent code is likely to execute on similar types of inputs and generate similar types of outputs. For example, different sorting algorithms often take the same array type as their input and output the same sorted array type. Code fragments with significantly different amount input and output variables are much less likely to be functionally equivalent.

Given a large set of code fragments, we first separate them into different groups according to the number of input and output variables they have—the fragments in the same group have the same numbers of input and output variables, then invoke Algorithm 4.2 on

every group in parallel. Alternatively, to prevent missing certain functionally equivalent code, such as those mentioned in Section 4.2.2, we can group the code fragments based on their estimated numbers of needed concrete values. In addition to increased degree of parallelism, another benefit of this grouping strategy is that we can generate inputs containing different numbers of concrete values for different groups so that groups with more input variables can have more values for increased testing accuracy.

4.4 Empirical Evaluation

This section presents our empirical experience with EqMiner on a benchmark program and the Linux kernel 2.6.24. The evaluations were carried out on a Fedora Core 5 system with a Xeon 3GHz CPU and 16GiB of memory, and a ROCKS computer cluster system with the SGE roll and varying numbers of hosts with an Opteron 2.6GHz CPU and 4GiB of memory.

4.4.1 Subject Programs

Sorting Benchmark We first used a benchmark program which contains several implementations of different sorting algorithms, including bubble sort, selection sort, recursive and non-recursive quick sort, recursive and non-recursive merge sort, and heap sort, to evaluate the effectiveness and accuracy of EqMiner.

Since the input generator in the prototype does not handle arrays, we wrote the benchmark in a way that the sorting algorithms accept a single struct containing a fixed number (seven) of integers as their input variable, but they internally cast the struct to an array before actual sorting. Also, the coexistence of recursive and non-recursive versions was used to evaluate the effects of the way EqMiner handles function calls (*cf.* Section 4.2.4).

There are about 350 lines of code in the program and 200 code fragments were generated when the *minStmtNum* was set to 10.

Linux Kernel 2.6.24. We used the Linux kernel as it is a large project with a relatively long history of development and a large number of participating developers, thus we can both evaluate the existence of functionally equivalent code in a popular software and test the scalability of our approach.

The Linux kernel 2.6.24 contains 9,730 C files totaling more than 6.2 million lines of code. Since our code chopper requires compilable code to obtain abstract syntax trees and control flow graphs, we only consider a subset of the kernel that is compilable on our Fedora Core 5 system. Specifically, we used the default setting when configuring the kernel before compilation, and saved the intermediate files (preprocessed C files with the suffix `.i`) for chopping.

We obtained 4,750 preprocessed C files which correspond to about 2.8 million lines of code in the original source code. Each file was a compilable unit and can be processed by the code chopper independently from other files. The total number of lines of code contained in the preprocessed files is not interesting due to the fact that these preprocessed files contained a lot of commonly used type definitions and function declarations and definitions. On the other hand, it is interesting to know the numbers of functions and statements contained in those functions since they directly affect the number of code fragments generated by the code chopper.

Among the 4,750 preprocessed files, CIL successfully parsed and constructed ASTs and CFGs for 3,748 files. There were 41,091 functions with unique names in the 3,748 preprocessed files, about $\frac{1}{3}$ of the total number of functions (more than 136K) in the Linux kernel [67]. If duplicated functions or functions with the same names in different files were also counted, the number would be more than 67K. We used these 67K functions for our following study and used the containing file and function name and line numbers as the unique identifier for each code fragment.

We calculated the numbers of primary statements contained in each function to estimate the total number of generated code fragments. Data showed that more than 26K functions contained fewer than 10 primary statement, while there were more than ten functions

contained more than 1,000 statements. Since we set the *minStmtNum* to 10, our code chopper ignored the code fragments for those small functions. Without respecting the statement boundaries, the code chopper generated more than 20 million code fragments due to the fact that more than ten functions contained thousands of statements. The number was reduced to about 6.5M when we respected boundaries.

4.4.2 Code Execution

An important decision we need to make is which code fragments we should use and how many test cases we should execute for each code fragment. Ideally, the more code fragments, the more functionally equivalent code we may find; the more test cases used, the more accurate the mined functionally equivalent code may be. On the other hand, even if each execution takes only one tenth of a second, it can take more than a week to sequentially execute each of the 6.5M code fragments once. In the following, we present several heuristic strategies we applied to address the scalability issue.

Code Fragment Sampling

Our code chopper in general generates quadratic number of code fragments *w.r.t.* the number of statements in a function. For example, a function named `serpent_setkey` has more than 1,600 sequential expression statements which led to more than 1.3M fragments for this function only. It appears uninteresting to consider *every* one of them for functional equivalence, but it is still interesting to consider *some* fragments that are "representative" and collectively cover significant portions of the function.

Our data showed that most functions (more than 85%) in the Linux kernel had fewer than 15 statements, implying most functions would have fewer than 120 code fragments generated. Thus, we again used random sampling: we randomly selected up to 100 code fragments from all the code fragments in each function to be used in our following study. With this strategy, the total number of code fragments that required further executions became 830,319, more than seven times smaller than the original 6.5M, but still covering

more than 1.6 million lines of code.[2]

Limiting Code Execution Time

Intuitively, most code fragments are small and their executions should finish quickly. On the other hand, there are code fragments that can fall into infinite loops if the randomly generated inputs do not satisfy certain requirements of the code. For example, one code fragment from the non-recursive merge sort in our benchmark looks like the following:

```
for(s=0; s < ArrayLength-b; s+=2*b) {
    ...
}
```

Since the variable `b` is identified as an input variable, random values will be fed into `b`. If `0` is used, the increment statement in the **`for`** loop will never change the value of `s` and cause an infinite loop.

Therefore, we imposed a limit on how long each execution of a code fragment can take. Preliminary evaluations on hundreds of fragments showed that if a fragment ever finished, it finished within 0.2 second. Thus, we set the limit to 0.5 second and killed the process if it exceeded that limit, and the output of the execution was marked as a failure with an error code for later output comparison. This strategy helped save tremendous amount of CPU time in our study.

Limiting the Number of Test Cases

Our study focused on functionally equivalent code, and any output difference between two pieces of code fragments would set them apart based on our code clustering algorithm. Assuming the input space of any code fragment, including invalid ones is uniformly sampled by our random input value generator, we have reason to believe if two code fragments have

[2]Note that, unlike the previous numbers for lines of code counted *w.r.t.* the kernel source files, this number was calculated *w.r.t.* preprocessed files for simplicity. Also, it may be better during the sampling process to ensure the selected code fragments accumulatively cover most statements in each function so that we may miss less functionally equivalent code.

the same outputs in ten out of *ten* test cases, they are likely to be functionally equivalent. On the other hand, even if uniform sampling is achieved, two code fragments may behave differently only on very rare cases and it may not be practical to distinguish one from another using random executions. This limitation is similar to that of traditional random testing [26, 139]. For example, the following two code fragments only differ at the **if** condition and they will only exhibit different behavior when `input` happens to be `23456`, which is very unlikely, even if we do not limit the range of random generated values (*cf.* Section 4.3).

```
if ( input < 23456 ) {
  ...
} else {
  ...
}
```

```
if ( input < 23457 ) {
  ...
} else {
  ...
}
```

Fortunately, we could consider such code fragments functionally *similar*, although not equivalent, since they only differ at rare cases. Section 4.5 discusses more about the concept of functional *similarity*.

Based on the above considerations, we set the limit for the number of test cases to 10. Thus, each code fragment would be executed at most $5! \times 10 = 1200$ times, taking at most 10 minutes.

The next section looks at some characteristics of the mined functionally equivalent code clusters.

4.4.3 Results of Functionally Equivalent Code Fragments

This section presents our findings on the sorting benchmark and the Linux kernel 2.6.24 and examines the accuracy of the results.

Sorting Benchmark

Within 104 minutes of sequential executions (no parallel executions at all), the 200 code fragments (no grouping at the beginning) were partitioned into 69 equivalence clusters. The

following summarizes the results of our inspection.

Most of the code fragments in the clusters are portions of the functions they belong to, instead of the whole functions. Their appearance in the same clusters are mainly due to two facts:

- Some portions of the different sorting algorithms are indeed functionally equivalent to each other, *e.g.*, portions of the recursive and non-recursive merge sort, and portions of bubble sort and selection sort.
- Some code fragments overlap with each other so much and there is no functional difference among them. While the second kind of functional equivalence is trivial, the first kind is more interesting since the existence of such functionally equivalence code fragments may indicate the need to extract commonly used code fragments for the purpose of reuse, which is one of the reasons why we carry out the study on functionally equivalent code.

At the level of whole functions, EqMiner correctly clustered the fragments that correspond to bubble sort, selection sort, non-recursive merge sort, and non-recursive quick sort into the same cluster. It was not surprising to see recursive merge sort and recursive quick sort were not in the cluster due to the current way EqMiner replaces function calls (*cf.* Section 4.2.4). For the heap sort, we noticed that a local variable in the function was identified as an output variable, which tricked the output comparison to view the code fragment differently: the local variable was defined as a flag that can affect the control flows of the code; it may not be used in one of the paths and was considered an output variable by our local reaching definition analysis (*cf.* Section 4.2.4). After we added a superficial statement that uses the local variable at the end of the heap sort function, the variable was no longer considered an output variable and the corresponding code fragment was then clustered with the code fragments for the other four algorithms. Alternatively, we used the strengthened definition of output variables (*cf.* Section 4.2.4), the above flag variable was no longer an output variable; however, it led to an opposite problem that some functionally

significant variables, *e.g.*, the variable storing the sorted data, were left out, causing both false positives and negatives.

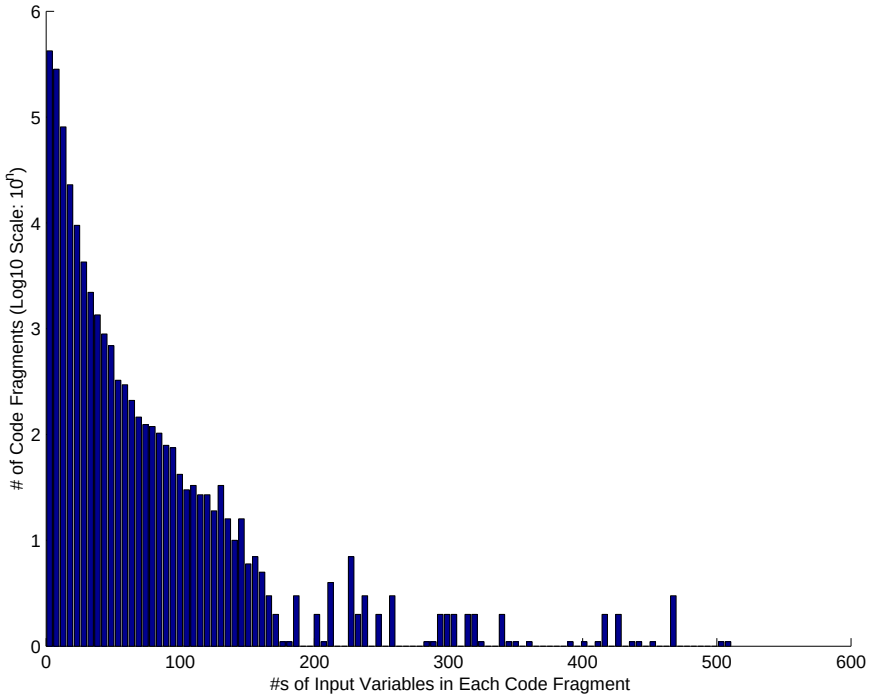
The evaluation based on the sorting algorithms showed the capability of EqMiner on mining functionally equivalent code fragments with satisfying accuracy. On the other hand, it showed the sensitivity of EqMiner on the automatically identified input and output variables. At the function level, it is often easy to improve the validity of the identified input and output variables based on the parameters of the function and its side effects and return values, but it is not obvious how to identify input and output variables for arbitrary portions of the function in general. The def-use analyses used in our approach is a reasonable semantic-aware heuristic, but it will still be worthwhile to investigate other heuristics in the future that can identify functionality-significant variables as inputs and outputs to help reduce both false positives and negatives.

Linux Kernel

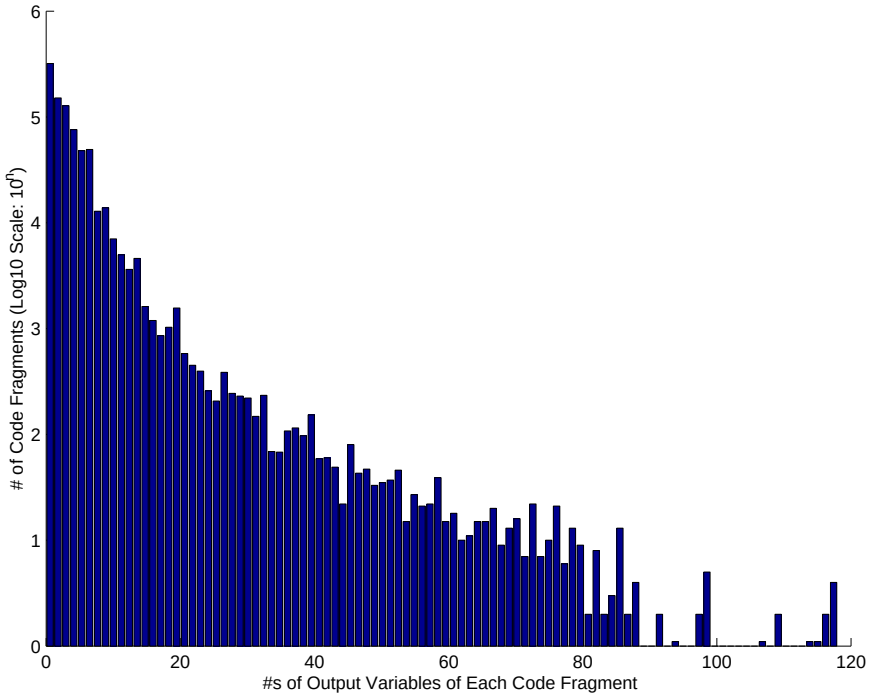
As mentioned in Section 4.4.2, 830,319 code fragments were used as candidates for functionally equivalent code.

Based on the numbers of input and output variables every candidate code fragment had, the code fragments were first separated into 2,909 groups of various sizes. The numbers of input variables in the code fragments ranged from 0 to 511, and the numbers of output variables ranged from 0 to 118. Figure 4.2 shows the histograms of the numbers of input and output variables in the code fragments respectively. We saw that most of the code fragments (51%) had fewer than six input variables so that our execution strategy on limiting the possible input permutations to 5! was reasonable.

The sizes of the groups (*i.e.*, the numbers of code fragments in the groups) ranged from 1 to 76972. The largest group contained code fragments that had two input variables and one output variable. Most groups (more than 90%) contained fewer than 200 code fragments, and more than 1,000 groups contained only one code fragment and there was no need to execute the code fragments in those groups. Also, only 18 groups (fewer than 1%) contained



(a) Semi-log histogram of the numbers of input variables



(b) Semi-log histogram of the numbers of output variables

Figure 4.2: Histograms for code fragments.

more than 10,000 code fragments. The data suggested that the time required to perform the executions for mining functionally equivalent code should be acceptable on our systems.

The executions of the code fragments in different groups were parallelized. We also parallelized the executions of the code fragments as mentioned at the end of Section 4.2.6 to speed up the clustering process on groups with large numbers of code fragments.

Since there were many users on our computer cluster system, the number of available hosts on the system varied significantly during the time period when we had our evaluations. Thus, the degree of parallelism was limited, ranging from several to 36 processes at a time. Also, the NFS server which we used to store all the code fragments and related data may have been a performance bottleneck that limited the actual degree of parallelism. We did not measure how many code executions were parallelized or how much CPU and disk time were. We simply recorded the wall clock time of all the executions, rounded to hours, and that was 189 hours, within 8 days.

The 830,319 code fragments were separated into 269,687 clusters. Most (164,994, more than 60%) of the clusters contained only one fragment, which means most fragments are not functionally equivalent to others. About 30% (82,907) of the clusters contained two to five fragments. Fewer than 1% (1,675) of the clusters contained more than 100 fragments. Many fragments in these large clusters actually overlapped with each other, and may be considered trivial. Also, many of these fragments came from static functions that were commonly included by many source files. For example, one cluster contained 33,225 fragments, most of which were generated from the same inline, static function `kmalloc` that was included in many preprocessed C files. Although such code fragments were trivially functionally equivalent, it showed the capability of EqMiner on detecting them. When we excluded such code fragments, only 159 non-overlapping fragments were left in the cluster.

Quantity of Functionally Equivalent Code We used a code filter to filter out all but one trivial code fragments in each cluster (which one to keep relies on the order of the occurrence of the overlapping fragments), and removed the cluster if only one code

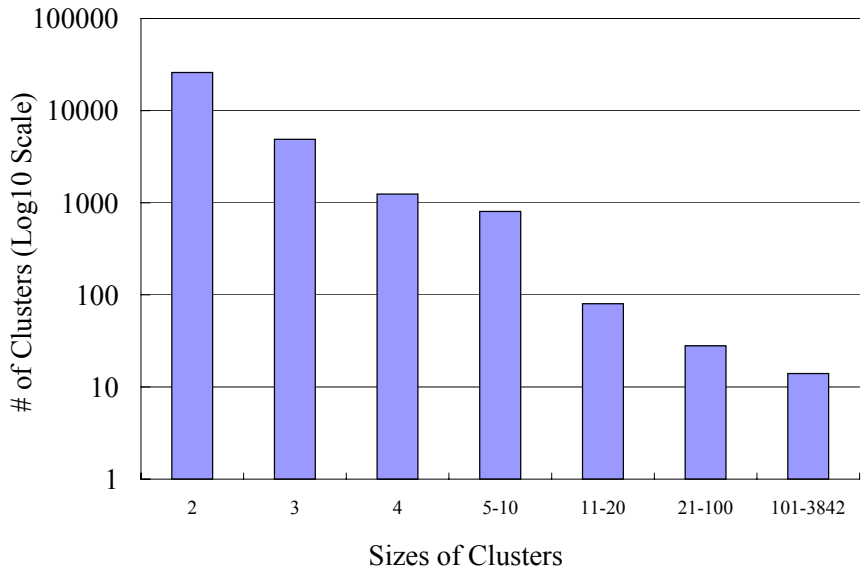


Figure 4.3: Histogram of the sizes of functionally equivalent clusters.

fragment was left in the cluster. We then obtained a set of 32,996 clusters that can be viewed as functional equivalence code clusters, covering about 624K lines of code in the Linux kernel. Figure 4.3 shows the histogram of the sizes of the clusters. Most of clusters (25,935) contained just two code fragments; very few (14) clusters contained more than 100 code fragments. On the other hand, there were still several clusters containing thousands of code fragments, and the largest one is 3,842.

The following code represents a common pattern of the code fragments in the largest cluster. A single output variable is identified, and it is assigned a value near the end of the code fragment through an input variable which is introduced due to various reasons, *e.g.*, a function call or an undefined variable.

```
output = 0;
... /* defs and uses of various variables */
output = input;
```

Assuming the input and output variables identified by EqMiner for these code fragments are appropriate, such code fragments are indeed functionally equivalent according to

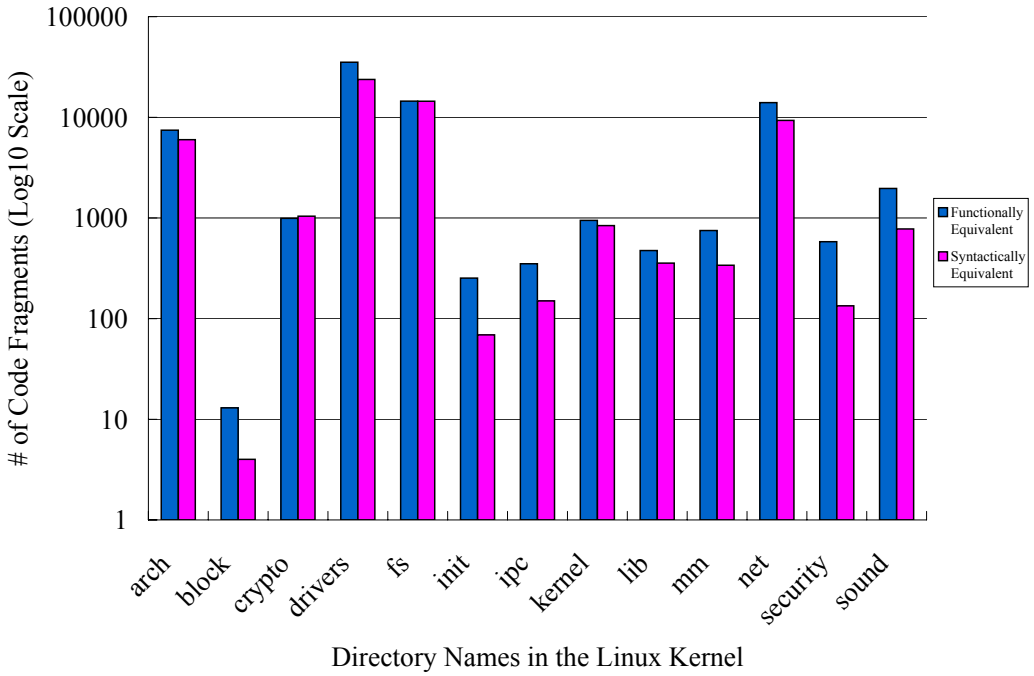


Figure 4.4: Spatial distribution of functionally and syntactically equivalent code in the Linux kernel.

our definition. However, whether it is really useful to consider them functionally equivalent is still a question worth of future investigation.

Figure 4.4 shows the spatial distribution of the mined functionally equivalent code in the Linux kernel directories. We can see that the `drivers` directory contains the most functionally equivalent code fragments (more than 35K), while the `block` directory contains the fewest (only 13). This distribution is similar to that of syntactically equivalent code fragments, which we discuss next.

Differences From Syntactically Equivalent Code Since many existing techniques can detect syntactically similar code, one may wonder what different results our approach can bring. Answering this question would also help to justify the significance of mining functionally equivalent code fragments in addition to syntactically similar ones. In the following, we present the result comparison between EqMiner and Deckard (Chapter 2).

As presented in Chapter 2, Deckard detects syntactically *similar* code by characteriz-

ing the syntax tree of a program as a set of vectors and searching for code fragments that have similar characteristic vectors. To have a common ground for comparison, we added the Line 15 in Algorithm 4.1 to generate a characteristic vector for any code fragment generated by EqMiner, and requested Deckard to use the set of vectors that corresponded to the 830,319 fragments in our study to search for syntactically *equivalent* code (*i.e.*, set its *Similarity* parameter to 1.0, but allow token level differences).

On one hand, the spatial distribution of syntactically equivalent code fragments detected by Deckard is similar to that of functionally equivalent ones (Figure 4.4). On the other hand, the two sets of code equivalence clusters are different in terms of the code fragments contained in the clusters.

We calculated the percentage of the code fragments that were contained in the clusters reported by EqMiner but not contained in the clusters reported by Deckard. To our surprise, the percentage was close to 91%, which means only 9% of the functionally equivalent code fragments were also syntactically equivalent. Notice that if there were overlapping code fragments in a cluster, all but one of them (often the first one in the clusters) were removed from the cluster by our code filter, while which one was the first may be different from cluster to cluster, causing the high number of unmatched code fragments between the two sets of clusters. After disabling the code filter, the difference percentage decreased to less than 58%, which means more than 42% functionally equivalent code fragments were syntactically equivalent. We also found that many code fragments reported by Deckard (more than 36%) were not reported by EqMiner, which means many syntactically equivalent code were functionally different. The still relatively large difference set between the two sets of clusters is an indication that the two kinds of code detection techniques, functionality-based and syntax-based, can complement each other.

Through a preliminary manual inspection on the difference sets of the clusters, we noticed several categories of the code fragments in the difference set that contributed to the differences:

- Many functionally equivalent code is indeed syntactically different. For example, the following two pieces of code are in fact functionally equivalent to "`output = input;`," an identity function that simply outputs its inputs.

```
output = 0;
while( output < input ) {
   ...
   output++;
}
```

```
if ( 0 ) {
   ...
} else {
   output = input;
}
```

- Lexical differences cause syntactically equivalent code to be functionally different. For example, DECKARD considers the code "`if(input < 10) output = 10;`" syntactically equivalent to the code "`if(input < 100) output = 100;`," while EQMINER considers them different from each other. As another example, the following pair of code fragments only differ at a single variable name and are syntactically equivalent, but they are functionally different.

```
output = 0;
if ( output < input ) {
   ...
   output = input + 1;
}
```

```
output = 0;
if ( output < input ) {
   ...
   output = output + 1;
}
```

- False positives produced by EQMINER may have contributed to a large portion of the difference set. For example, DECKARD recognizes function calls and considers calls with different numbers of parameters syntactically different, but EQMINER treats any function call as a random input variable and thus may report false functionally equivalent code fragments.

In the following paragraphs, we discuss the accuracy of EQMINER further.

Accuracy As we previously presented, we limited the number of test cases executed for each code fragment to 10. This restriction helped improve the performance of EQMINER,

but it may cause false positives in the sense that it may incorrectly put functionally different code fragments in the same cluster.

We used additional test cases to evaluate the accuracy of EqMiner. Two different measurements, one stricter than the other, were used to calculate false positive (FP) rates:

First false positive rates: Given a cluster C, its *first false positive rate* $\mathcal{R}_1(C)$ is the number of the code fragments in C that have different outputs from its representative's outputs during the additional testing over the number of all code fragments in C. As a special case, if the former number is just one smaller than the latter, *i.e.*, no code fragments in the cluster are functionally equivalent, we increase the former number by one and thus let $\mathcal{R}_1(C)$ be 100%.

Given a set of clusters $\mathbb{C}$, its *first false positive rate* $\mathcal{R}_1(\mathbb{C})$ is the number of all such code fragments in $\mathbb{C}$ that have different outputs from its corresponding representative's outputs during the additional testing over the total number of all code fragments in $\mathbb{C}$. The special cases when no code fragments in a cluster are functionally equivalent are handled in a way similar to the above.

Second false positive rates: Given a cluster C, its *second false positive rate* $\mathcal{R}_2(C)$ is the number of singleton clusters generated during the additional testing (*i.e.*, the number of code fragments considered functionally nonequivalent to any other code fragments in C) over the number of all code fragments in C.

Given a set of clusters $\mathbb{C}$, its *second false positive rate* $\mathcal{R}_2(\mathbb{C})$ is the number of all such code fragments in $\mathbb{C}$ that are put into singleton clusters during the additional testing over the total number of all code fragments in $\mathbb{C}$.

The first false positive rate is more strict in the sense that it tells how many code fragments in a cluster may not belong to *the* cluster, while the second false positive rate tells how many code fragments in a cluster may not belong to *any* functionally equivalent cluster.

Ideally, we should carry out additional tests for every cluster. Due to the limitation of computing resources, we only focused on 128 clusters each of which contains fewer than 100 code fragments: we randomly selected 50 clusters from the clusters sized between 2 and 4, another 50 clusters from the clusters sized between 5 and 20, and included all of the clusters (28) sized between 20 and 100. We did not choose the clusters in a uniformly random way since the sizes of the clusters are not distributed uniformly (*cf.* Figure 4.3) and we were trying not to spend too much time on large clusters. The set of clusters we chose contain $1,913$ code fragments, and we denote the whole set as E. We then allowed each code fragment from E to execute with 100 randomly generated inputs.

To compute the first false positive rate $\mathcal{R}_1(E)$, we first executed the representative in every cluster in E with 100 random inputs and recorded their outputs for each of the inputs. Then, we executed all other code fragments in E with the same inputs in parallel. For each code fragment, whenever it generated an output different from the corresponding output of its representative, it was marked as a false positive and its execution was terminated. In addition, if all code fragments in a cluster except for the representative were marked, the representative was also marked as a false positive. This process was finished in about 13 hours on our cluster system, and we obtained the false positive rate $\mathcal{R}_1(E) = 28\%$.

To compute the second false positive rate $\mathcal{R}_2(E)$, we used a different strategy from the above: we simply invoked Algorithm 4.2 on each of the cluster in E, in parallel, with the number of test cases set to 100. The process was finished in about 16 hours on our cluster system. Although the number of clusters increased to 206 from 128, only 57 code fragments were in singleton clusters, which means the false positive rate $\mathcal{R}_2(E)$ is within 3%.

In addition, we excluded the false positives marked during the calculation of $\mathcal{R}_1(E)$ from E and executed an additional 100 random tests on the rest of E in order to further justify the false positive rates. We noticed that there were additional 69 code fragments marked as false positives, increasing the false positive rate $\mathcal{R}_1(E)$ to 32%. Also, we invoked Algorithm 4.2 on the resulting clusters (excluding singleton clusters) generated during the calculation of the previous $\mathcal{R}_2(E)$ with additional 100 tests, and we only noticed 6 new

singleton clusters, increasing the false positive rate $\mathcal{R}_2(E)$ to 3.5%.

On one hand, the relatively low second false positive rate showed that the code fragments mined by EqMiner were very likely to functionally equivalent to some others. On the other hand, we noticed that several factors may have contributed to the relatively high first false positive rates, including implementation limitations, invalid inputs, and inadequate test coverage. Further investigation may help to decide whether directed random testing techniques that combine concrete and symbolic executions [72, 161] can help alleviate the problem of generating valid, sufficient tests that also exists in traditional random testing [26, 139].

4.5 Discussions and Future Work

As discussed in our earlier publication [94], there is still room for improvements, from the refinement of the definition of functional equivalence to more robust implementations.

Scalability As described in Section 4.4.2, EqMiner employed various heuristics to reduce the expenses of computing resources. Some of the heuristics, such as limiting the number of code fragments by sampling, may lead to missed functionally equivalent code (*i.e.*, false negatives), while others, such as limiting the number of test cases, may increase false positive rates. It will be a challenging and interesting task to scale EqMiner to as many code fragments as possible with as many test cases as possible. It may require the combination of novel techniques and significant engineering efforts to simplify the problem or explore the degree of parallelism in this problem further. Existing program analysis techniques may be of some help. For example, directed testing that combines symbolic and concrete executions [72, 161] may help reduce the number of test cases required to explore the execution paths and functionality of each code fragment, thus reduce expenses on code executions without increasing false positive rates. Also, program slicing, either static or dynamic [17,76,180], may help our code chopper to focus on the most relevant code portions

and reduce the number of candidate code fragments.

Code Chopping In this dissertation, we generate code fragments mainly based on the syntax of a sequence of statements. In fact, many syntactically consecutive statements may not be semantically related to each other, as shown in [67] that more than half of the functions in the Linux kernel perform more than one unrelated computations. It is intuitively uninteresting to put statements for different computations in the same code fragment and consider it a candidate for functionally equivalent ones. Thus, utilizing program dependency information and generating code fragments based on program slices may help to exclude uninteresting candidates and leave with us more semantic-relevant ones for further consideration. Also, since program slices are often smaller than a whole function body, the number of code fragments generated by the new code chopper would be smaller and help scale up EqMiner.

Identifying Input and Output Variables Section 4.4.3 has discussed that EqMiner can be sensitive to the input and output variables identified for each code fragment. The liveness and reaching definition analyses used in EqMiner may include functionally insignificant variables in the sets of input and output variables, causing false positives and negatives. If the code chopping was carried out on program slices, such mis-identifications could be fewer since input and output variables are often more prominent and meaningful along the data flows within slices. Other heuristics, such as statistical learning, may leverage programmers' knowledge and help to identify more appropriate variables as inputs and outputs.

Functional Equivalence Definition This chapter defines functional equivalence based on the same input and output behavior, which is different from the classic concept of *semantic equivalence* based on program semantics, such as operational semantics. We in effect do not consider intermediate program states in our definition and have not attempted to detect semantically equivalent code yet. As a result, our approach may not be directly

applicable for plagiarism detection, for example, among student programming homeworks. In principle, we can use functional equivalence to search for semantically equivalent code: first identify the smallest units of code (*e.g.*, a statement) that are functionally equivalent to some others, then look for compositions of such code units that are consecutive and still functionally equivalent to some others. Repeated compositions of consecutive code units may thus form larger code fragments that are semantically equivalent. It would be future work to investigate the feasibility and complexity of such a problem.

We have not explored the concept of functional *similarity* in the sense that we only considered code fragments that are equivalent and have not considered code fragments that are equivalent on certain inputs but different on others. It would be ideal to have a general definition for similarity so that functional differences between code fragments may be quantified and studied further. For example, we may say the following pair of code has a similarity 0.8 since they behave differently on two out of ten inputs, supposing the input domain is the whole set of integers:

```
if ( input % 10 == 0 ) {
   output = 0;
} else {
   output = 1;
}
```

```
if ( input % 10 == 1 ) {
   output = 0;
} else {
   output = 1;
}
```

Broader Applicability Although our implementation is only for C language, the definition of functional equivalence is only concerned with the input and output behavior of arbitrary piece of code and not binded to any particular programming language. This general concept of functional equivalence would be applicable to not only source code, but also binary code. Further, detecting code clones across different languages of different abstraction levels would also be possible. This would potentially enable more applications such as cross-language code and component reuses.

Categorization and Application Categorizing functionally equivalent code fragments may help us to understand the characteristics of the code and understand further about how equivalent code occurs and evolves. One immediate application of such a study will be functionality-based refactoring that helps extract functionally equivalent code into shared libraries for easy reuse. It will be a valuable complement for syntax-based refactoring in the current mainstream. Also, our study can enable semantic-aware code search, in addition to syntax-based search approaches, that may help improve developer productivity. In addition, small functional differences among similar code may be useful for detecting program errors, similar to many other types of syntactic or semantic inconsistencies that have been used for bug detection [53, 58, 95].

We only performed limited investigation on the mined code clusters, and have not derived general knowledge about the patterns or characteristics of the code clusters. Future work will investigate further in this direction, and aim to categorize the characteristics of mined functionally equivalent code fragments, to increase the accuracy of the code clones, and to explore their potential applications.

Chapter 5

Related Work

This chapter discusses closely related studies on code clones and related code detection and analysis techniques, and classifies them into the categories.

5.1 Similarity Detection

This dissertation focuses on defining, detecting, and analyzing source code similarity. Many notions and techniques presented in the dissertation may also be applicable to data beyond source code and relate to previous work on those domains.

5.1.1 Source Code Clone Detection

As mentioned in Section 1.2, there have been quite a number of detection techniques that target diffeernt kinds of code clones classified in the spectrum in Figure 1.1.

String-Based Approaches A program is first divided into strings, usually lines. Each code fragment consists of a contiguous sequence of strings. Two code fragments are similar if their constituent strings match. The representative work here is Baker's "parameterized" matching algorithm, *Dup* ([9, 10]), where identifiers and literals are replaced with a global constant to normalize strings with minor differences and help to detect more similar code.

Token-Based Approaches A program is processed to produce a token sequence, which is scanned for duplicated token subsequences that indicate potential code clones. Compared to string-based approaches, a token-based approach is usually more robust against code changes (*e.g.*, comments and spacing). CCFinder [101] and CP-Miner [119] are perhaps the most well-known among token-based techniques, where CCFinder employes a suffix-tree algorithm [78] to find similar token subsequences and CP-Miner uses a frequent subsequence mining algorithm in data mining, *CloSpan* [4,182], to find repeatedly occurring subsequences even with gaps. Some software plagiarism detection tools (*e.g.*, Moss [155] and JPlag [145]) also use token-based techniques to search for similar code fragments.

Tree-Based Approaches A program is parsed to produce a parse tree or abstract syntax tree (AST) representation of the source program. Exact or close matches of subtrees can then be identified by comparing subtrees within the generated parse tree or AST [18, 19, 61, 112, 166, 174]. Alternatively, different metrics can be used to *fingerprint* the subtrees, and subtrees with similar fingerprints are reported as possible duplicates [111, 127]. DECKARD is also tree-based, but because of our novel use of characteristic vectors and efficient vector clustering techniques, it detects significantly more clones and is much more scalable.

Birthmark-Based Approaches A program is often fingerprinted in particular ways and fingerprints for different pieces of code are checked against each other to find similar code. The approaches are often used for the purposes of detecting plagiarism and protecting software intellectual properties. Various kinds of fingerprints have been proposed for detecting illegal theft code or code clones, either static ones that are based on the program's source code lines, tokens, syntax trees, control flow graphs, or dependency graphs [40, 86, 96, 127, 155], or dynamic ones that are based on program execution traces and states [44, 157], or combined ones [45, 191]. Although these techniques are often aware of certain program semantics, they are sensitive to the defined fingerprints and not made scalable, and which fingerprint is more appropriate may depend on a particular application.

Graph-Based Approaches Approaches that take some semantic information (*e.g.*, data and control dependencies) into consideration have also been proposed. Komondoor and Horwitz [110] suggest the use of program dependence graphs (PDGs) [62] and program slicing [175] to find isomorphic PDG subgraphs in order to identify code clones. They also propose an approach to group identified clones together while preserving the semantics of the original code [109] for automatic procedure extraction to support software refactoring. Liu *et al.* [123] apply a relaxed subgraph isomorphism algorithm to look for plagiarised code. These techniques are more robust in dealing with code formatting than tree-based approaches, but they have not scaled to large code bases, while our work has proposed a general framework to make both tree-based and graph-based approaches much more scalable.

The existence of diversified code clone detection techniques and tools also calls for comparisons. Bellon *et al.* [22] present an experiment conducted in 2002 that evaluates six clone detectors (CCFinder [101, 124], CLAN [111, 115, 127], CloneDR [18, 19], Dup [10, 11], Duplix [114, 128], and Duploc [57, 151]) in terms of recall and precision as well as space and time expenses, and concludes that a different tools have different strength and weaknesses and thus may be suitable for different contexts and applications. Rysselberghe and Demeyer [153] compare three detection techniques and conclude that different techniques may be useful for different scenarios. There are also many other tools (*e.g.*, Clone Digger [33], KClone [91], CloneDetective [100], PMD/CPD [1], SDD [118], *etc.*) that came into existence in recent years and have not been included in the comparison. An up-to-date, large-scale comparison among the state-of-the-art tools would be an insightful experiment to perform in the near future. The mutation/injection based framework proposed by Roy and Cory [152] may be used to automatically generate several kinds of code clones and compare precisions and recalls of different tools.

5.1.2 Similarity Detection on More General Data Structures

Various techniques for finding data of similar structures are also becoming popular. For example, data of tree-structures (*e.g.*, XML databases) and similarity detection on such data are gaining increasing attention. However, efficient tree similarity detection still remains an open problem, while similarity detection on high dimension numerical vectors has already been extensively studied and efficient algorithms exist. Yang *et al.* [183] propose an approximation algorithm for computing tree editing distances based on q-level vectors. Similar algorithms exist for searching approximately same strings based on characteristic vectors for strings q-grams (*i.e.*, all strings of length q over a fixed alphabet) [103, 171]. We adapt their characterization to capture structural information in parse trees, and apply LSH [50] to search for similar trees. To the best of our knowledge, DECKARD is by far the most scalable and effective tool for tree-based clone detection.

For data of graph-like structures, graph or subgraph isomorphism algorithms are abundant. Although subgraph isomorphism is a NP-Complete problem and the complexity for graph isomorphism even remains unresolved [69], there exist many approximate algorithms [27,52,73]. Krinke *et al.* [114] use paths of limited lengths in graphs to approximate the program dependence graphs and look for code clones by looking for matching paths. We have also worked on PDGs for code clone detection and our approach maps graphs back to syntax trees and is much more scalable than previous work.

Similarity detection techniques are also applicable to binary code, in addition to source code. Schulman [158] fingerprints binary code and applies a string matching algorithm to find similar binary code. Sæbjørnsen *et al.* [154] generate characteristic vectors for disassembled binary code and also use LSH to find binary clones. Related to binary similarities, security communities have many studies on malware detection which looks for binaries with certain matching signatures or behaviors [20,41,42,184]. There would be abundant research opportunities to explore the connection between clone detection and malware detection.

5.1.3 Higher-Level Clone Detection

Clones also occur due to similarities at levels higher than source code, such as design patterns, software architectures, software models. Many clone detection techniques may be applied to detect such high-level clones.

Basit and Jarzabek [15,16] employ an algorithm for frequent itemset mining [74] to find higher-level syntactic clones among a set of source code clones found by CCFinder [101]. Pham *et al.* [142] utilize characteristic vectors defined for graphs [135] to detect model-level clones in Matlab/Simulink models.

Ishio *et al.* [87] define a set of rules to transform Java source code into sequences and apply the PrefixSpan algorithm [141] to find crosscutting concerns. Bruntink *et al.* [30,31] propose clone class metrics to extract aspects from code-level clones. Shi and Olsson [163] rediscover design patterns from Java programs by analyzing code structures and system behavior. PR-Miner [120] also uses frequent itemset mining to detect implicit, high-level programming patterns for specification discovery or bug detection. For example, "an allocator a must be followed by a deallocator b" is an example of such high-level similarities. Ammons *et al.* [5] apply machine learning approach to infer program specifications by observing program execution traces and summarizing the frequent API interaction patterns as state machines. Lo *et al.* [125] propose a specification mining architecture with trace filtering and clustering techniques to improve the accuracy, robustness and scalability of specification miners. Kremenek *et al.* [113] expressed program properties as annotation variables and then inferred the annotations by encoding various types of evidence, including domain specific knowledge about a property, on Annotation Factor Graphs. Gabel *et al.* [68] use BDD [32] based techniques to learn simple generic patterns and compose them into large, complex specifications, providing a fully automated, trace-based API miner. Although our approach currently operates at the code level, the detection algorithms based syntax trees and program dependence graphs can also be used to detect higher-level clones as long as we adjust vector generation to appropriately model corresponding problems. We leave for

future work the application of our algorithms on such high-level pattern discovery tasks.

5.2 Studies on Code Clones

As introduced in Chapter 1, the purpose of detecting code clones is to enable many important applications. Much work has also studies detected clones in different ways to facilitate different applications.

5.2.1 Clone Refactoring

A few independent studies address the question of how much clone coverage in large open-source projects. The goal is to determine what fraction of a program is duplicated code and how much it affects code complexity and maintenance cost. It is difficult to directly compare these studies because such results are usually sensitive to many factors: the different definitions of code similarity used, the particular detection algorithms used, the various choices of parameters for these algorithms, and the different code bases used for evaluation (*e.g.*, CCFinder [82, 101] reports 29% cloned code in JDK, and CP-Miner [119] reports 22.7% cloned code in Linux kernel 2.6.6). However, these studies do confirm that there is a significant amount of duplicated code in large code bases.

Many code refactoring techniques and tools based on code clones are also proposed to improve code readability, quality, and maintainability. Baxter *et al.* [18] extract and refactor code clones as macros so that programs can be rewritten in more concise forms. Balazinska *et al.* [12] present a clone classification scheme for assessing and measuring different system reengineering opportunities. They also extract clone differences and interpret them in terms of programming language entities to support object-oriented refactoring [13]. The proposed classification considers each group of cloned Jarzabek *et al.* [90] use meta-programming techniques to reduce code redundancies. They also introduce XVCL, a variant configuration language, to allow flexible reuse of generic, adaptable meta-components [89]. Yu and Ramaswamy [186] classify automatically detected code clones into different categories and

propose strategies to refactor the clones to improve modularity. Yoshida *et al.* [185] propose a more semantic-aware refactoring technique and tool based on dependencies among clones (also called "chained clones"). Tairas and Gray [167] use Latent Semantic Indexing (LSI) to detect trends and associations among clone clusters and determine if they provide further comprehension to assist in the maintenance of clones. They also propose a domain-specific language for representing code clones in order to perform analysis and suggest refactoring opportunities on the clones [169].

Although people often believe that duplicated code is harmful to the quality of code and causes additional maintenance cost, we must acknowledge that some clones exist for good reasons, as mentioned in Section 3.1. In particular, Rajapakse *et al.* [148] suggest that unifying clones may not be always desirable because of its impact on system qualities and performance. Kapser and Godfrey [102] also identify several patterns of cloning that may even be beneficial to code quality, and suggest that code refactoring that reduces the amount of code clones may not always be the best solution; instead, tools that can help to maintain multiple instances of a clone group synchronously (*e.g.*, Linked Editing presented by Toomim *et al.* [170]) may be a better option. CloneTracker [55, 56] use abstract clone region descriptor to track changes in clone groups in evolving software and provide notifications and assistances for developers to modify clones consistently. CReN [88] provides identifier tracking and renaming supports for code clones in integrated development environments, such as Eclipse. Hou *et al.* [84] further identify several design elements for supporting code clone tracking and management.

5.2.2 Clone Evolution

Related to clone tracking and management mentioned above, the need for such techniques is not just because we cannot always refactor clones away, it is also because clones themselves keep changing along with the evolution of software from one version to another. Research has studied how clones in software projects are introduced or removed over time across different versions, aiming to understand the evolution and dynamics of clones.

Laguë *et al.* [115] examined six versions of a telecommunication software system and found that a significant number of clones were removed due to refactoring, but the overall number of clones increased due to the faster rate of clone introduction. Antoniol *et al.* [6] model clones across versions of a program as time series, and use the predictive model to study the evolution of clones. Kim *et al.* [104] describe a study of clone genealogies, using CCFinder as the clone detector. They find that many code clones are short-lived and thus performing aggressive refactoring may not be worthwhile, and that long-lived clones pose great challenges to refactoring because they evolve independently and can deviate significantly from the original copy.

In addition, the need for incremental clone detection is also emerging due to fasting evolving software. It is often more costly to run clone detection and analysis on a whole set of code than on a changed set of code from one version to another. A number of techniques, in addition to the above ones that track clones along code changes, have been proposed to address the problem recently. Gode and Koschke [71] detect clones based on the detection results for the previous version and create mappings among clones of different versions to supply information about additions and deletions of clones. Such incremental analysis can be cheap and useful for on-the-fly detection and evolutionary clone analysis. Nguyen *et al.* [136] propose a framework for clone management that constructs clone groups and tracks and updates clone groups when code evolves. They also incorporate the capability of clone management into software configuration management systems (SCMs, *e.g.*, Subversion) to make SCMs clone-aware [137].

With the increasing popularity of open source software, open source code repositories are also becoming abundant. Much information related to open source software development, including code changes, bug tracks, email archives, developer migrations, *etc.* is readily available. Such information can help to answer many interesting research questions, *e.g.*, how does software change [192], what is the social structure among open source developers [25], and what are the effects of distributed development processes on software quality [149]. It would also valuable future work to use such data to answer some clone-

specific questions as well, *e.g.*, how do clones evolve along the development process, how do clones affect code quality, and what are the correlations between software processes and clones.

5.2.3 Clone Visualization

To aid studies on clone patterns and their evolutions, many tools have also been developed to visualize software and code clones in different ways. Johnson [97] tries to use graphs (*e.g.*, Hasse diagrams) to provide insights into the structure of code redundancies in the GNU GCC compiler. CCFinder [101,124] uses scatter plots and heat maps to show the locations and densities of clones. Tairas *et al.* [168] look into an alternative visualization method by extending the AspectJ Development Tool Visualiser as an Eclipse plugin to visualize the clone results from a free version of CloneDR [19]. SoftGUESS [3] provides users with a mechanism to interactively explore clone structures both through direct manipulation as well as a domain-specific language. Chevalier *et al.* [39] visualize the evolution of the code clone structure by emphasizing both changing and constant code patterns.

5.2.4 Bug Detection

Studies have also proposed bug detection techniques based on the general observation as ours that inconsistencies can be indications of bugs. The concept is applicable not only to code clones, but also higher-level clones mentioned in Section 5.1.3.

As shown in Section 3.4.2, both CP-Miner [119] and our approch look for inconsistencies at the source code level; the difference is that we look in the contexts surroudning clones, in addition to the clones themselves. Engler *et al.* [58] and PR-Miner [120] aim to detect violations of programming rules which are mined from programs themselves, which are in spirit close to much of Engler's follow-up work on Metal [8, 79]; a difference is that inconsistencies are mainly automatically discovered in PR-Miner and our work, but need to be given in Metal-related work. Ammons *et al.* [5] and Kremenek *et al.* [113] consider the problem in the context of program specifications: they look for inconsistencies among

detected program specifications for bugs. Also related is Xie *et al.*'s work [179] on using redundancies in programs, such as idempotent operations, unused values, dead code, un-taken conditional branches, and redundant null-checks, to flag possible bugs. Dillig *et al.* [53] look for inconsistent uses of the same pointer to find null-pointer dereference errors.

The technique used in EqMiner to generate code fragments, although simpler and more straightforward, can be viewed as a specific instance of the general notions of program slicing [175] and data slicing [37,181]. Similar techniques, which simplify or remove parts of a program that have no effect on a concerned property, have been applied to bug detection. *Delta debugging* [43, 129, 188] is an example; it provides an automated way to simplify failure-inducing inputs and helps to locate failure-related program states. Gupta *et al.* [77] use the intersection of forward and backward program slices to reduce the sizes of code for further debugging. Zhang *et al.* [190] use value profiles of statements to prune statements in dynamic slies that are unlikely related to program failures. Sterling and Olsson propose the concept of *program chipping* [165] to automatically remove parts of a program so that the part that contributes to some symptomatic output becomes more apparent. Their tool, ChipperJ, is an implementation for Java programs. Similar to EqMiner, ChipperJ works on syntax trees and has to deal with code compilation and the problem of invalid executions (*e.g.*, infinite loops).

In a broader sense, our clone-related bug detection is also related to the large body of work on bug detection techniques, including software model checking techniques (*e.g.*, SLAM/BLAST [14,81] and Saturn [177,178]), type inference and checking (*e.g.*,CQual [64, 65], Vault [51], and Osprey [93]), dataflow analysis-based approaches (*e.g.*, ESP [49]), and verification-based approaches (*e.g.*, ESC/Java [63] and LCLint [60]). These techniques have been applied in various settings to find software errors in large systems [35, 54, 98, 116, 162, 173]. Most of these techniques require programmer annotations. Our detection of clone-related bugs do not require any annotations, and many of the errors detected by our technique cannot be detected by these techniques. On the other hand, these systems are generally sound (*i.e.*, they can show the absence of certain classes of bugs), while ours is

not and may have both high false positives and false negatives. detection. As discussed in Section 3.5, we believe that our approach complements well these existing techniques.

Besides static techniques, there are also dynamic bug detection techniques and combinations of the two kinds. Brun and Ernst [29] use Daikon, a dynamic invariant detection tool [59], to generate properties of programs, and use machine learning algorithms (*e.g.*, Decision Trees [130], Support Vector Machines [34], Random Forests [28]) to identify fault-revealing properties based on similarities and differences among different versions of the same program properties are fault-revealing or not. DIDUCE [80] also tracks dynamic program invariants (similar to Daikon) and discovers abnormal behavior by examining instructed program runs. These are dynamic approaches, while our work on detecing clone-related bugs is static.

5.3 Program Equivalence

Code clone detection, especially finding functionally equivalent code, is closely related to the classic problem of program equivalence [47], which is undecidable in general. Definitions of program equivalence based on operational semantics have been proposed long time before [143, 150]. Definitions based on input and output behavior have also been investigated in the literature [23, 48, 187]. However, to the best of our knowledge, our work is the first that uses random testing for *large-scale* detection of functionally equivalent *code fragments*. Previous work mostly considers equivalence among programs or functions, instead of arbitrary code fragments, and requires clear input and output interface for the programs or functions under consideration. Also, previous work focuses more on checking equivalence between two given pieces of code, instead of detecting equivalence among a large number of pieces of code, partly due to scalability constraints.

The most closely related work on using random testing for program equivalence is Podgurski and Pierce's *behavior sampling* [144]. They identify functionally relevant routines by executing candidates on a set of random or user-specified inputs and comparing their

outputs to user-provided outputs. There are several significant technical differences between their work and ours which may complement each other:

- Their technique was proposed mainly to *query* for a routine that satisfies certain requirements from an existing library, for the purpose of reuse and program synthesis, while our technique is mainly to *detect* functionally equivalent code from a large set of sources and may be used to construct a library containing reusable code.

- Their technique works for routines where inputs and outputs are well defined and requires a one-to-one correspondence (including variable number and types) between the inputs of candidate routines and those of user-specified inputs, while our technique works for arbitrary code fragments and allows input and output variables to have different types.

- Our subject program is of millions of lines of code which is more challenging in terms of scalability.

- Their technique also considers behavior sampling for abstract data types (ADTs) [122] since they consider types as part of their functional equivalence definition, while we consider primitive input values without types. Many ideas and challenges mentioned in their paper, such as handling code with complex interfaces and facilitating automatic program synthesis based on reusable code libraries and such code search capability, are similar to ours and remain as intriguing future work.

5.4 Random Testing

Although working on different granularities, our technique used for detecting functionally equivalent code in fact shares a similar property with any software testing activity [21], such as regression testing, that aims to uncover functional differences among programs and specifications: it guarantees functional differences when it separates code fragments into different clusters, but it cannot fully guarantee functional equivalence.

As we utilize random testing in this dissertation, our techniques also share the same challenges faced with random testing [26, 140]:

- The probability of generating particular inputs that cause particular program behavior may be very small and it is very difficult to generate sufficient inputs that expose all program behaviors (*a.k.a.* test coverage problem).
- Many random generated input values may lead to the same observable behavior and are thus redundant.

As we mentioned in Section 4.5, the test coverage problem may be addressed by introducing the concept of *functional similarity*: if we have a way to quantify behavioral differences between two arbitrary programs (*e.g.*, measuring "sizes" of unexposed behaviors of a program against another, as Offutt *et al.* [139] measure "sizes" of program faults), unexposed program behaviors may be modeled by a similarity threshold and no longer require concrete inputs to test.

Symbolic executions [24, 172, 176], combined with concrete executions [36, 72, 117, 161], can also be applied to increase test coverage and reduce test redundancy. Similar techniques may be adapted for our setting to generate the most cost-efficient inputs to effectively expose different program behaviors for the clone detection purpose.

Chapter 6

Conclusions

This dissertation has presented techniques for scalable and accurate detection of similar code and their applications. These techniques are applicable to a wide spectrum of similarity definitions, from syntax-based ones to functionality-based ones. Since similar code commonly occurs in large software projects, detecting, tracking, and managing similar code play an important role in improving software quality, reducing maintenance cost, and increasing development productivity.

6.1 Summary

We have presented a spectrum of code clones based on their semantic-awareness. The spectrum provides a unified view of most existing code clone detection techniques and tools and may help identify appropriate requirements when applying clone detection and analysis to different applications.

We have proposed a general framework for reducing complex similarity problems to vector similarity problems in the Euclidean space with efficient clustering schemes available. Within the framework, code clones based on parse trees, abstract syntax trees, control flow graphs, or program dependency graphs can all be detected by clustering characteristic vectors for these structures. More generally, the idea of generating characteristic vectors as

a first step for detecting similar code has broader applicability, including but not limited to clones in binary code, clones in different programming languages, and even clones in articles in natural languages, as long as we can find a feasible way to extract characteristic vectors that preserve application-specific features of the original code or articles.

Empirical evaluations have showed that the general framework easily scales to millions of lines of code with few false positives, not only for syntax trees but also for graphs. In addition, it is language-agnostic and can be easily parallelized; it thus has the potential to scale to the billions of lines of existing open source software.

We have also expanded our efforts into more semantic-aware techniques. We have particularly developed a random testing based approach for finding functionally equivalent code fragments. It is the first of its kind to explore functionally equivalent code in a large scale (the Linux kernel with millions of lines of code). Our empirical study has showed that many functionally equivalent but syntactically different code fragments exist and may help expand the applicable domains of previous clone-based techniques and applications.

As a sample application, we have presented a novel application of clone-based bug detection. In particular, we have proposed a general notion of *context-based inconsistencies* as indicators of code clone-related bugs and presented three concrete types of such inconsistencies. Then, the concepts have been applied to the clones identified by DECKARD and many previously unknown bugs in the Linux kernel and Eclipse have been discovered. These bugs exhibited diverse characteristics and are difficult to detect with any single previous bug detection technique.

6.2 Outlook

In each of the previous chapters, we have discussed some short-term future work which is related to different aspects of this dissertation and concerned more about specific technical details and improvements. Here, we consider longer-term directions for code clone detection and analysis.

Promoting Search-Assisted Development Paradigm

As introduced in Chapter 1, one significant application of code clone detection and analysis is to facilitate code reuse, likely through an easily reusable library containing commonly used code clones. To construct a library that fully utilizes the available open source software which may be in different programming languages and collectively reach trillions of lines of code, we need code clone detection and analysis techniques and tools that can scale, work across language boundaries, extract clone patterns, generate code summaries, and aggregate common usage patterns. In addition, we need techniques that work not just at the code level, but also at higher levels, such as design patterns and software architectures, as well as at meta-levels, such as error-proneness and time/space complexity of each piece of code. High-level and meta-level characteristics will help developers to understand and reuse existing code more efficiently, and help them to prevent similar errors from happening again.

Following the construction of such a large, reusable library, *search-assisted programming*, where a code search engine is integrated with development environments, would become an obvious necessity. The code search engine, built upon efficient code clone detection and analysis technqiues, should be able to answer many of developers' programming questions, ranging from syntax to semantics, from simple code reuse to overall design choices. Here are some sample questions that should be easy for the search engine to find answers from the library:

- What is the standard way to use a particular API?
- How should code perform a common task, *e.g.*, sending a file through the HTTP protocol with a particular set of APIs?
- Which design pattern is best suited for this particular context?
- How should these particular components be organized and reused?

In addition to programming, the library and the code search engine can also be extended to include test cases and debugging information to help enable *search-assisted testing and debugging*. Then, developers could also reuse tests and debugging knowledge by asking suitable questions. For example, are bugs that previously occurred under similar contexts also possible in this context? Is there a known test case for similar code usable for this code? Is there any known defect that has similar symptoms as this one? Is there a better way to make this code use less memory?

Although this *search-assisted development paradigm* (combining programming, testing, and debugging) may still be far from realization, it is a realistic goal that aims to utilize all kinds of information about existing code to help developers to make more intelligent development choices and fewer errors. Similar to token-level word auto-completion provided by existing development environments, a foreseeable feature of the paradigm is to provide automatic code change suggestions and validations based on its surrounding contexts, such as a better way to use a given API, the correctness of the way the API is used, the complexity of a given piece of code, and a better optimized version of the code.

Facilitating Interdisciplinary Research

Techniques presented in this dissertation have their connections to many areas in computer science, such as program analysis, software testing, computational geometry, and information retrieval. For example, the idea of generating characteristic vectors and clustering in the framework proposed in DECKARD is similar in spirit to indexing and clustering ideas in the area of information retrieval. If we look at the problem of code clone detection and analysis abstractly, it is indeed similar to the problem of information retrieval which aims to find certain documents within or information about documents. Much research on data mining and machine learning can also be applied to the problem of code clone detection. On the other hand, code is not exactly the same as normal texts; its particular structures and meanings may require different mining techniques other than normal information retrieval techniques and may boost the advance of other areas. Program analysis techniques can

help to reveal such structures and much hidden properties of code so as to facilitate more effective clone detection and analysis.

Expanding Applicable Domains

Subject programs used in the empirical studies in this dissertation are more standalone ones, but we expect the techniques presented in this dissertation are also applicable to many different kinds of software, such as binary code as we mentioned before, web applications which often involve multi-layer, multi-language programming, and embedded software which is often closely tied to particular physical devices. With the evolving programming languages and software development practices, the details of particular clone detection and analysis techniques may also keep evolving; however, we believe research on code clones will remain active across all software applications due to the ubiquitous existence of clones.

A Central Goal

As a final note, a central goal of research on code clones is to improve software quality, increase development productivity, and reduce development and maintenance cost. Studying clones is to reduce unnecessary duplication, facilitate formation of common code libraries, improve code reusability, drive search-and-reuse based program synthesis and development, and promote high-productivity, high-understandability programming styles. Much exciting code clone-related work remains to be done towards the goal of achieving high-quality, low-cost software, and the success of such work will rely on how well we can distill and organize useful information from large existing code bases and incorporate such information into software engineering processes.

Bibliography

[1] *Finding Duplicate Code with PMD/CPD*. http://pmd.sourceforge.net/cpd.html.

[2] *The Stanford Parser: A Statistical Parser*. http://nlp.stanford.edu/software/lex-parser.shtml.

[3] Eytan Adar and Miryung Kim. SoftGUESS: Visualization and exploration of code clones in context. In *ICSE'07: Proceedings of the 29th International Conference on Software Engineering*, pages 762–766, Minneapolis, Minnesota, USA, May 20–26, 2007. IEEE Computer Society.

[4] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *ICDE'95: Proceedings of the 11th International Conference on Data Engineering*, pages 3–14, Taipei, Taiwan, March 6–10, 1995. IEEE Computer Society.

[5] Glenn Ammons, Rastislav Bodik, and James R. Larus. Mining specification. In *POPL'02: Proceedings of the 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–6, Portland, Oregon, USA, January 16–18, 2002. ACM.

[6] Giuliano Antoniol, Gerardo Casazza, Massimiliano Di Penta, and Ettore Merlo. Modeling clones evolution through time series. In *ICSM'01: Proceedings of the 2001 IEEE International Conference on Software Maintenance*, pages 273–280, Florence, Italy, November 6–10, 2001. IEEE Computer Society.

[7] Vikraman Arvind and Piyush P. Kurur. Graph isomorphism is in SPP. *Information and Computation*, 204(5):835–852, May 2006. Elsevier.

[8] Ken Ashcraft and Dawson Engler. Using programmer-written compiler extensions to catch security holes. In *S&P 2002: Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 143–159, Berkeley, California, USA, May 12–15, 2002. IEEE Computer Society.

[9] Brenda S. Baker. On finding duplication and near-duplication in large software systems. In *WCRE'95: Proceedings of the 2nd Working Conference on Reverse Engineering*, pages 86–95, Toronto, Canada, July 14–16, 1995. IEEE Computer Society.

[10] Brenda S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM Journal on Computing (SICOMP)*, 26(5):1343–1362, October 1997. Society for Industrial and Applied Mathematics.

[11] Brenda S. Baker. Finding clones with Dup: Analysis of an experiment. *IEEE Transactions on Software Engineering (TSE)*, 33(9):608–621, September 2007. IEEE Computer Society.

[12] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lague, and Kostas Kontogiannis. Measuring clone based reengineering opportunities. In *METRICS'99: Proceedings of the 6th International Software Metrics Symposium*, pages 292–303, Boca Raton, Florida, USA, November 4–6 1999. IEEE Computer Society.

[13] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lagüe, and Kostas Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *WCRE'00: Proceedings of the 7th Working Conference on Reverse Engineering*, pages 98–107, Brisbane, Queensland, Australia, November 23–25, 2000. IEEE Computer Society.

[14] Thomas Ball and Sriram K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL'02: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–3, Portland, Oregon, USA, January 16–18, 2002. ACM.

[15] Hamid Abdul Basit and Stan Jarzabek. Detecting higher-level similarity patterns in programs. In *ESEC/FSE'05: Proceedings of the 5th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 156–165, Lisbon, Portugal, September 5–9, 2005. ACM.

[16] Hamid Abdul Basit and Stan Jarzabek. A data mining approach for detecting higher-level clones in software. *IEEE Transactions on Software Engineering (TSE)*, 35(4):497–514, July–August 2009. IEEE Computer Society.

[17] Samual Bates and Susan Horwitz. Incremental program testing using program dependence graphs. In *POPL'93: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 384–396, Charleston, South Carolina, USA, January 1993. ACM.

[18] Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. DMS®: Program transformations for practical scalable software evolution. In *ICSE'04: Proceedings of the 26th International Conference on Software Engineering*, pages 625–634, Edinburgh, Scotland, UK, May 23–28, 2004. IEEE Computer Society.

[19] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *ICSM'98: Proceedings of the 1998 IEEE International Conference on Software Maintenance*, pages 368–377, Bethesda, Maryland, USA, November 16–19, 1998. IEEE Computer Society.

[20] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Krügel, and Engin Kirda. Scalable, behavior-based malware clustering. In *NDSS'09: Proceedings*

of the 16th Annual Network and Distributed System Security Symposium, San Diego, California, USA, February 8–11, 2009. The Internet Society (ISOC).

[21] Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold Co., New York, USA, 2nd edition, 1990.

[22] Stefan Bellon, Rainer Koschke, Giuliano Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering (TSE)*, 33(9):577–591, September 2007. IEEE Computer Society.

[23] Miquel Bertran, Francesc-Xavier Babot, and August Climent. An input/output semantics for distributed program equivalence reasoning. *Electronic Notes in Theoretical Computer Science*, 137(1):25–46, July 2005. Elsevier.

[24] Dirk Beyer, Adam J. Chlipala, and Rupak Majumdar. Generating tests from counterexamples. In *ICSE'04: Proceedings of the 26th International Conference on Software Engineering*, pages 326–335, Edinburgh, Scotland, UK, May 23–28, 2004. IEEE Computer Society.

[25] Christian Bird, David Pattison, Raissa D'Souza, Vladimir Filkov, and Premkumar Devanbu. Latent social structure in open source projects. In *FSE'08: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 24–35, Atlanta, Georgia, USA, November 9–14, 2008. ACM.

[26] David L. Bird and Carlos Urias Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3):229–245, 1983. IBM.

[27] Hans L. Boblaender. Polynomial algorithms for graph isomorphism and chromatic index on partial k-trees. *Journal of Algorithms*, 11(4):631–643, December 1990. Elsevier.

[28] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, October 2001. Springer.

[29] Yuriy Brun and Michael D. Ernst. Finding latent code errors via machine learning over program executions. In *ICSE'04: Proceedings of the 26th International Conference on Software Engineering*, pages 480–490, Edinburgh, Scotland, UK, May 23–28, 2004. IEEE Computer Society.

[30] Magiel Bruntink. Aspect mining using clone class metrics. In *WARE'04: Proceedings of the 1st Workshop on Aspect Reverse Engineering, co-located with WCRE 2004*, Delft, The Netherlands, November 9th, 2004. Published as CWI Technical Report SEN-E0502, February 2005.

[31] Magiel Bruntink, Arie van Deursen, Remco van Engelen, and Tom Tourwé. On the use of clone detection for identifying crosscutting concern code. *IEEE Transactions on Software Engineering (TSE)*, 31(10):804–818, October 2005. IEEE Computer Society.

[32] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers (TC)*, 35(8):677–691, August 1986. IEEE Computer Society.

[33] Peter Bulychev and Marius Minea. Duplicate code detection using anti-unification. In *SYRCoSE 2008: Proceedings of 2008 Spring Young Researchers' Colloquium on Software Engineering*, volume 2, pages 51–54, Saint-Petersburg, Russia, May 29–30, 2008. Institute for System Programming of the Russian Academy of Sciences (ISP/RAS).

[34] Christopher J. C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2(2):121–167, June 1998. Springer.

[35] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience (SP&E)*, 30(7):775–802, June 2000. John Wiley & Sons, Inc.

[36] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. In *CCS'06: Proceedings of the 13th ACM Conference on Computer and Communications Security*, pages 322–335, Alexandria, Virginia, USA, October 30 – November 3, 2006. ACM.

[37] Tat W. Chan and Arun Lakhotia. Debugging program failure exhibited by voluminous data. *Journal of Software Maintenance: Research and Practice*, 10(2):111–150, March–April 1998. John Wiley & Sons, Inc.

[38] Liming Chen and Algirdas Avizienis. *N*-version programming: A fault-tolerance approach to reliability of software operation. In *FTCS-25: Proceedings of the 25th International Symposium on Fault-Tolerant Computing, "Highlights from Twenty-Five Years"*, volume 3, pages 113–119, Pasadena, California, USA, June 27–30 1995. IEEE Computer Society.

[39] Fanny Chevalier, David Auber, and Alexandru Telea. Structural analysis and visualization of C++ code evolution using syntax trees. In *IWPSE'07: Proceedings of the 9th International Workshop on Principles of Software Evolution, held in conjunction with the 6th Joint Meeting of ESEC/FSE*, pages 90–97, Dubrovnik, Croatia, September 3–4, 2007. ACM.

[40] Michel Chilowicz, Etienne Duris, and Gilles Roussel. Syntax tree fingerprinting for source code similarity detection. In *ICPC'09: Proceedings of the 17th IEEE International Conference on Program Comprehension, co-located with ICSE 2009*, Vancouver, British Columbia, Canada, May 17–19, 2009. IEEE Computer Society.

[41] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. In *SEC'03: Proceedings of the 12th USENIX Security Symposium*, pages 169–186, Washington, District of Columbia, USA, August 4–8, 2003. USENIX Association.

[42] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Xiaodong Song, and Randal E. Bryant. Semantics-aware malware detection. In *S&P 2005: Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 32–46, Oakland, California, USA, May 8–11, 2005. IEEE Computer Society.

[43] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *ICSE'05: Proceedings of the 27th International Conference on Software Engineering*, pages 342–351, St. Louis, Missouri, USA, May 15–21, 2005. ACM.

[44] Christian S. Collberg, Edward Carter, Saumya K. Debray, Andrew Huntwork, John D. Kececioglu, Cullen Linn, and Michael Stepp. Dynamic path-based software watermarking. In *PLDI'04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 107–118, Washington, District of Columbia, USA, June 9–11, 2004. ACM.

[45] Christian S. Collberg and Clark D. Thomborson. Software watermarking: Models and dynamic embeddings. In *POPL'99: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 311–324, San Antonio, Texas, USA, January 20–22, 1999. ACM.

[46] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, Massachusetts, USA, 2nd edition, September 2001.

[47] Guy Cousineau and Patrice Enjalbert. Program equivalence and provability. In *MFCS'79: Proceedings of the 8th Symposium on Mathematical Foundations of Computer Science*, volume 74 of *Lecture Notes in Computer Science (LNCS)*, pages 237–245, Olomouc, Czechoslovakia, September 3–7, 1979. Springer.

[48] Roy L. Crole and Andrew D. Gordon. A sound metalogical semantics for input/output effects. In *CSL'94: Selected Papers from the 8th International Workshop on Computer*

Science Logic, 1994, volume 933 of *Lecture Notes in Computer Science (LNCS)*, pages 339–353, Kazimierz, Poland, September 25–30, 1995. Springer.

[49] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI'02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 57–68, Berlin, Germany, June 17–19, 2002. ACM.

[50] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *SoCG'04: Proceedings of the 20th ACM Symposium on Computational Geometry*, pages 253–262, Brooklyn, New York, USA, June 8–11, 2004. ACM.

[51] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI'01: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 59–69, Snowbird, Utah, USA, June 20–22, 2001. ACM.

[52] Fred DePiero and David Krout. An algorithm using length-r paths to approximate subgraph isomorphism. *Pattern Recognition Letters*, 24(1–3):33–46, January 2003. Elsevier.

[53] Isil Dillig, Thomas Dillig, and Alex Aiken. Static error detection using semantic inconsistency inference. In *PLDI'07: Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 435–445, San Diego, California, USA, June 10–13, 2007. ACM.

[54] Nurit Dor, Michael Rodeh, and Shmuel Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *PLDI'03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 155–167, San Diego, California, USA, June 9–11, 2003. ACM.

[55] Ekwa Duala-Ekoko and Martin P. Robillard. Tracking code clones in evolving software. In *ICSE'07: Proceedings of the 29th International Conference on Software Engineering*, pages 158–167, Minneapolis, Minnesota, USA, May 20–26, 2007. IEEE Computer Society.

[56] Ekwa Duala-Ekoko and Martin P. Robillard. CloneTracker: Tool support for code clone management. In *ICSE'08: Proceedings of the 30th International Conference on Software Engineering*, pages 843–846, Leipzig, Germany, May 10–18, 2008. ACM.

[57] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *ICSM'99: Proceedings of the 1999 IEEE International Conference on Software Maintenance*, pages 109–118, Oxford, England, UK, August 30 – September 3, 1999. IEEE Computer Society.

[58] Dawson R. Engler, David Yu Chen, and Andy Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *SOSP'01: Proceedings of the 18th ACM Symposium on Operating System Principles*, pages 57–72, Banff, Alberta, Canada, October 21–24, 2001. ACM.

[59] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *ICSE'00: Proceedings of the 22nd International Conference on Software Engineering*, pages 449–458, Limerick, Ireland, June 4–11, 2000. ACM.

[60] David Evans, John Guttag, James Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. In *FSE'94: Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 87–96, New Orleans, Louisiana, USA, December 6–9, 1994. ACM.

[61] Raimar Falke, Pierre Frenzel, and Rainer Koschke. Empirical evaluation of clone detection using syntax suffix trees. *Empirical Software Engineering*, 13(6):601–643, December 2008. Springer.

[62] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, July 1987. ACM.

[63] Cormac Flanagan, K.Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI'02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 234–245, Berlin, Germany, June 17–19, 2002. ACM.

[64] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *PLDI'99: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 192–203, Atlanta, Georgia, USA, May 1–4, 1999. ACM.

[65] Jeffrey S. Foster, Tachio Terauchi, and Alexander Aiken. Flow-sensitive type qualifiers. In *PLDI'02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 1–12, Berlin, Germany, June 17–19, 2002. ACM.

[66] Free Software Foundation. *GNU Diffutils for Comparing and Merging Files*. http://www.gnu.org/software/diffutils/manual/.

[67] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *ICSE'08: Proceedings of the 30th International Conference on Software Engineering*, pages 321–330, Leipzig, Germany, May 10–18, 2008. ACM.

[68] Mark Gabel and Zhendong Su. Javert: Fully automatic mining of general temporal properties from dynamic traces. In *FSE'08: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 339–349, Atlanta, Georgia, USA, November 9–14, 2008. ACM.

[69] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, San Francisco, California, USA, January 1979.

[70] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *VLDB'99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 518–529, Edinburgh, Scotland, September 7–10, 1999. Morgan Kaufmann.

[71] Nils Göde and Rainer Koschke. Incremental clone detection. In *CSMR'09: Proceedings of the 13th European Conference on Software Maintenance and Reengineering*, pages 219–228, Fraunhofer IESE, Kaiserslautern, Germany, March 24–27, 2009. IEEE Computer Society.

[72] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *PLDI'05: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 213–223, Chicago, Illinois, USA, June 12–15, 2005. ACM.

[73] Marco Gori, Marco Maggini, and Lorenzo Sarti. Exact and approximate graph matching using random walks. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 27(7):1100–1111, July 2005. IEEE Computer Society.

[74] Gösta Grahne and Jianfei Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *FIMI'03: Proceedings of the 1st IEEE ICDM Workshop on Frequent Itemset Mining Implementations*, volume 90, Melbourne, Florida, USA, November 19, 2003. CEUR Workshop Proceedings (CEUR-WS.org). Also as RPI Technical Report 03-04.

[75] GrammaTech. *CodeSurfer*. http://www.grammatech.com/.

[76] Alex Groce and Rajeev Joshi. Exploiting traces in program analysis. In *TACAS'06: Proceedings of the 12th International Conference on Tools and Algorithms for the*

Construction and Analysis of Systems, held as part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2006, volume 3920 of *Lecture Notes in Computer Science (LNCS)*, pages 379–393, Vienna, Austria, March 25 – April 2, 2006. Springer.

[77] Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. Locating faulty code using failure-inducing chops. In *ASE'05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 263–272, Long Beach, CA, USA, November 7–11, 2005. ACM.

[78] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, Cambridge, England; New York, USA, 1st edition, May 1997.

[79] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *PLDI'02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 69–82, Berlin, Germany, June 17–19, 2002. ACM.

[80] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE'02: Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, Orlando, Florida, USA, May 19–25, 2002. ACM.

[81] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *POPL'02: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 58–70, Portland, Oregon, USA, January 16–18, 2002. ACM.

[82] Yoshiki Higo. *Code Clone Analysis Methods for Efficient Software Maintenance*. PhD thesis, Osaka University, Japan, 2006.

[83] Susan Horwitz and Thomas W. Reps. The use of program dependence graphs in software engineering. In *ICSE'92: Proceedings of the 14th International Conference on Software Engineering*, pages 392–411, Melbourne, Australia, May 11–15, 1992. ACM.

[84] Daqing Hou, Patricia Jablonski, and Ferosh Jacob. CnP: Towards an environment for the proactive management of copy-and-paste programming. In *ICPC'09: Proceedings of the 17th IEEE International Conference on Program Comprehension, co-located with ICSE 2009*, Vancouver, British Columbia, Canada, May 17–19, 2009. IEEE Computer Society.

[85] Watts S. Humphrey. A personal commitment to software quality. In *ESEC'95: Proceedings of the 5th European Software Engineering Conference*, volume 989 of *Lecture Notes in Computer Science (LNCS)*, pages 5–7, Sitges, Spain, September 25–28, 1995. Springer.

[86] Hyun-il Lim, Heewan Park, Seokwoo Choi, and Taisook Han. Detecting theft of Java applications via a static birthmark based on weighted stack patterns. *IEICE Transactions on Information and Systems*, E91–D(9):2323–2332, September 2008. Institute of Electronics, Information and Communication Engineers.

[87] Takashi Ishio, Hironori Date, Tatsuya Miyake, and Katsuro Inoue. Mining coding patterns to detect crosscutting concerns in Java programs. In *WCRE'08: Proceedings of the 15th Working Conference on Reverse Engineering*, pages 123–132, Antwerp, Belgium, October 15–18, 2008. IEEE Computer Society.

[88] Patricia Jablonski and Daqing Hou. CReN: A tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE. In *ETX'07: Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange*, pages 16–20, Montréal, Québec, Canada, October 21, 2007. ACM.

[89] Stan Jarzabek, Paul Bassett, Hongyu Zhang, and Weishan Zhang. XVCL: XML-based variant configuration language. In *ICSE'03: Proceedings of the 25th International Conference on Software Engineering*, pages 810–811, Portland, Oregon, USA, May 3–10, 2003. IEEE Computer Society.

[90] Stan Jarzabek and Shubiao Li. Eliminating redundancies with a "composition with adaptation" meta-programming technique. In *ESEC/FSE'03: Proceedings of the 4th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 237–246, Helsinki, Finland, September 1–5, 2003. ACM.

[91] Yue Jia, Dave Binkley, Mark Harman, Jens Krinke, and Makoto Matsushita. KClone: A proposed approach to fast precise code clone detection. In *IWSC'09: Proceedings of the 3rd International Workshop on Software Clones, held in conjunction with CSMR 2009 (the 13th European Conference on Software Maintenance and Reengineering)*, Fraunhofer IESE, Kaiserslautern, Germany, March 24, 2009. IEEE Computer Society.

[92] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stéphane Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *ICSE'07: Proceedings of the 29th International Conference on Software Engineering*, pages 96–105, Minneapolis, Minnesota, USA, May 20–26, 2007. IEEE Computer Society.

[93] Lingxiao Jiang and Zhendong Su. Osprey: A practical type system for validating dimensional unit correctness of C programs. In *ICSE'06: Proceedings of the 28th International Conference on Software Engineering*, pages 262–271, Shanghai, China, May 20–28, 2006. ACM.

[94] Lingxiao Jiang and Zhendong Su. Automatic mining of functionally equivalent code fragments via random testing. In *ISSTA'09: Proceedings of the 18th International Symposium on Software Testing and Analysis*, pages 81–92, Chicago, Illinois, USA, July 19–23, 2009. ACM.

[95] Lingxiao Jiang, Zhendong Su, and Edwin Chiu. Context-based detection of clone-related bugs. In *ESEC/FSE'07: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 55–64, Dubrovnik, Croatia, September 3–7, 2007. ACM.

[96] Howard J. Johnson. Identifying redundancy in source code using fingerprints. In *CASCON'93: Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research*, volume 1, pages 171–183, Toronto, Ontario, Canada, October 24–28, 1993. IBM.

[97] Howard J. Johnson. Visualizing textual redundancy in legacy source. In *CASCON'94: Proceedings of the 1994 Conference of the Centre for Advanced Studies on Collaborative Research*, page 32, Toronto, Ontario, Canada, October 31 – November 3, 1994. IBM.

[98] Robert Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *SEC'04: Proceedings of the 13th USENIX Security Symposium*, pages 119–134, San Diego, California, USA, August 9–13, 2004. USENIX Association.

[99] Capers Jones. *Estimating Software Costs: Bringing Realism to Estimating*. McGraw-Hill Companies, New York, USA, 2nd edition, April 2007.

[100] Elmar Juergens, Florian Deissenboeck, and Benjamin Hummel. CloneDetective – A workbench for clone detection research. In *ICSE'09: Proceedings of the 31st International Conference on Software Engineering*, pages 603–606, Vancouver, British Columbia, Canada, May 16–24, 2009. IEEE Computer Society.

[101] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering (TSE)*, 28(7):654–670, July 2002. IEEE Computer Society.

[102] Cory Kapser and Michael W. Godfrey. "Cloning considered harmful" considered harmful. In *WCRE'06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 19–28, Benevento, Italy, October 23–27, 2006. IEEE Computer Society.

[103] Juha Kärkkäinen. Computing the threshold for q-gram filters. In *SWAT'02: Proceedings of the 8th Scandinavian Workshop on Algorithm Theory*, volume 2368 of *Lecture Notes In Computer Science (LNCS)*, pages 348–357, Turku, Finland, July 3–5, 2002. Springer.

[104] Miryung Kim, Vibha Sazawal, and David Notkin. An empirical study of code clone genealogies. In *ESEC/FSE'05: Proceedings of the 5th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 187–196, Lisbon, Portugal, September 5–9, 2005. ACM.

[105] Dan Klein and Christopher D. Manning. Fast exact inference with a factored model for natural language parsing. In *Advances in Neural Information Processing Systems 15 (NIPS 2002)*, pages 3–10, Vancouver, British Columbia, Canada, December 9–14, 2002. MIT Press 2003.

[106] Dan Klein and Christopher D. Manning. Accurate unlexicalized parsing. In *ACL'03: Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics*, pages 423–430, Sapporo, Japan, July 7–12, 2003. ACL.

[107] John C. Knight and Nancy G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering (TSE)*, 12(1):96–109, January 1986. IEEE Computer Society.

[108] Donald Ervin Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, 3rd edition, July 1997.

[109] Raghavan Komondoor and Susan Horwitz. Semantics-preserving procedure extraction. In *POPL'00: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 155–169, Boston, Massachusetts, USA, January 19–21, 2000. ACM.

[110] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *SAS'01: Proceedings of the 8th International Static Analysis Symposium*, volume 2126 of *Lecture Notes in Computer Science (LNCS)*, pages 40–56, Paris, France, July 16–18, 2001. Springer.

[111] Kostas Kontogiannis, Renato de Mori, Ettore Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Automated Software Engineering*, 3(1–2):77–108, July 1996. Springer.

[112] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *WCRE'06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 253–262, Benevento, Italy, October 23–27, 2006. IEEE Computer Society.

[113] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. From uncertainty to belief: Inferring the specification within. In *OSDI'06: Proceedings of the 7th Symposium on Operating System Design and Implementation*, pages 161–176, Seattle, Washington, USA, November 6–8, 2006. USENIX Association.

[114] Jens Krinke. Identifying similar code with program dependence graphs. In *WCRE'01: Proceedings of the 8th Working Conference on Reverse Engineering*, pages 301–309, Stuttgart, Germany, October 2–5, 2001. IEEE Computer Society.

[115] Bruno Laguë, Daniel Proulx, Jean Mayrand, Ettore Merlo, and John P. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *ICSM'97: Proceedings of the 1997 IEEE International Conference on*

Software Maintenance, pages 314–321, Bari, Italy, October 1–3, 1997. IEEE Computer Society.

[116] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *SEC'01: Proceedings of the 10th USENIX Security Symposium*, pages 177–190, Washington, District of Columbia, USA, August 13–17, 2001. USENIX Association.

[117] Eric Larson and Todd Austin. High coverage detection of input-related security faculties. In *SEC'03: Proceedings of the 12th USENIX Security Symposium*, pages 121–136, Washington, District of Columbia, USA, August 4–8, 2003. USENIX Association.

[118] Seunghak Lee and Iryoung Jeong. SDD: High performance code clone detection system for large scale source code. In *OOPSLA'05: Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 140–141, San Diego, California, USA, October 16–20, 2005. ACM.

[119] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *OSDI'04: Proceedings of the 6th Symposium on Operating System Design and Implementation*, pages 289–302, San Francisco, California, USA, December 6–8, 2004. USENIX Association.

[120] Zhenmin Li and Yuanyuan Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE'05: Proceedings of the 5th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 306–315, Lisbon, Portugal, September 5–9, 2005. ACM.

[121] Soyini Liburd. An *n*-version electronic voting system. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2004.

[122] Barbara Liskov and Stephen Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*, pages 50–59, Santa Monica, California, USA, April 1974. ACM.

[123] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. GPLAG: Detection of software plagiarism by program dependence graph analysis. In *KDD'06: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 872–881, Philadelphia, Pennsylvania, USA, August 20–23, 2006. ACM.

[124] Simone Livieri, Yoshiki Higo, Makoto Matsushita, and Katsuro Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder. In *ICSE'07: Proceedings of the 29th International Conference on Software Engineering*, pages 106–115, Minneapolis, Minnesota, USA, May 20–26, 2007. IEEE Computer Society.

[125] David Lo and Siau-Cheng Khoo. SMArTIC: Towards building an accurate, robust and scalable specification miner. In *FSE'06: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 265–275, Portland, Oregon, USA, November 5–11, 2006. ACM.

[126] David Mandelin, Lin Xu, Rastislav Bod'ik, and Doug Kimelman. Jungloid mining: Helping to navigate the API jungle. In *PLDI'05: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 48–61, Chicago, Illinois, USA, June 12–15, 2005. ACM.

[127] Jean Mayrand, Claude Leblanc, and Ettore Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *ICSM'96: Pro-*

ceedings of the 1996 IEEE International Conference on Software Maintenance, pages 244–254, Monterey, California, USA, November 4–8, 1996. IEEE Computer Society.

[128] Ettore Merlo, Giulio Antoniol, and Jens Krinke. Identifying similar code with metrics and program dependence graphs. Unpublished manuscript.

[129] Ghassan Misherghi and Zhendong Su. Hdd: Hierarchical delta debugging. In *ICSE'06: Proceedings of the 28th International Conference on Software Engineering*, pages 142–151, Shanghai, China, May 20–28, 2006. ACM.

[130] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, USA, March 1997.

[131] Lajos Nagy, Richard Ford, and William Allen. *N*-version programming for the detection of zero-day exploit. In *IEEE Topical Conference on Cybersecurity*, Daytona Beach, Florida, USA, April 2006.

[132] National Institute of Standards and Technology (NIST). *Software Errors Cost U.S. Economy* $59.5 *Billion Annually*. http://www.nist.gov/public_affairs/releases/n02-10.htm, June 28, 2002.

[133] George C. Necula, Scott Mcpeak, S. P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC'02: Proceedings of the 11th International Conference on Compiler Construction, held as part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2002*, volume 2304 of *Lecture Notes in Computer Science (LNCS)*, pages 213–228, Grenoble, France, April 8–12, 2002. Springer.

[134] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI'07: Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 89–100, San Diego, California, USA, June 10–13, 2007. ACM.

[135] Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Accurate and efficient structural characteristic feature extraction for clone detection. In *FASE'09: Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering, held as part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2009*, volume 5503 of *Lecture Notes in Computer Science (LNCS)*, pages 440–455, York, UK, March 22–29, 2009. Springer.

[136] Tung Nguyen, Hoan Nguyen, Jafar Al-Kofahi, Nam Pham, and Tien Nguyen. Scalable and incremental clone detection for evolving software. In *ICSM'09: Proceedings of the 25th IEEE International Conference on Software Maintenance*, Edmonton, Alberta, Canada, September 20–26, 2009. IEEE Computer Society.

[137] Tung Nguyen, Hoan Nguyen, Nam Pham, Jafar Al-Kofahi, and Tien Nguyen. Clone-aware configuration management. In *ASE'09: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, Auckland, New Zealand, November 16–20, 2009. ACM.

[138] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, January 1999.

[139] A. Jefferson Offutt and J. Huffman Hayes. A semantic model of program faults. In *ISSTA'96: Proceedings of the 1996 International Symposium on Software Testing and Analysis*, volume 21 of *SIGSOFT Software Engineering Notes*, pages 195–200, San Diego, CA, USA, January 8–10, 1996. ACM.

[140] Carlos Pacheco and Michael D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP'05: Proceedings of the 19th European Conference on Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science (LNCS)*, pages 504–527, Glasgow, Scotland, July 27–29, 2005. Springer.

[141] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *ICDE'01: Proceedings of the 17th International Conference on Data Engineering*, pages 215–224, Heidelberg, Germany, April 2–6, 2001. IEEE Computer Society.

[142] Nam H. Pham, Hoan Anh Nguyen, Tung Thanh Nguyen, Jafar M. Al-Kofahi, and Tien N. Nguyen. Complete and accurate clone detection in graph-based models. In *ICSE'09: Proceedings of the 31st International Conference on Software Engineering*, pages 276–286, Vancouver, British Columbia, Canada, May 16–24, 2009. IEEE Computer Society.

[143] Andrew M. Pitts. Operational semantics and program equivalence. In *Applied Semantics: Advanced Lectures, International Summer School (APPSEM 2000)*, volume 2395 of *Lecture Notes in Computer Science (LNCS), Tutorial*, pages 378–412. Springer, Caminha, Portugal, September 9–15, 2002.

[144] Andy Podgurski and Lynn Pierce. Retrieving reusable software by sampling behavior. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2(3):286–303, July 1993. ACM.

[145] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science*, 8(11):1016–1038, November 2002. A Publication of Graz University of Technology and Universiti Malaysia Sarawak, in cooperation with Know-Center and Campus02.

[146] Daniel Quinlan, Markus Schordan, Qing Yi, and Andreas Sæbjørnsen. Classification and utilization of abstractions for optimization. In *ISoLA'04: Proceedings of the 1st International Symposium on Leveraging Applications of Formal Methods*, pages 57–73, Paphos, Cyprus, October 30–November 2, 2004. Springer.

[147] Daniel J. Quinlan. *ROSE: An Open Source Compiler Infrastructure*. Lawrence Livermore National Laboratory (LLNL), http://www.rosecompiler.org/.

[148] Damith C. Rajapakse and Stan Jarzabek. Using server pages to unify clones in web applications: A trade-off analysis. In *ICSE'07: Proceedings of the 29th International Conference on Software Engineering*, pages 116–126, Minneapolis, Minnesota, USA, May 20–26, 2007. IEEE Computer Society.

[149] Narayan Ramasubbu and Rajesh Krishna Balan. Globally distributed software development project performance: An empirical analysis. In *ESEC/FSE'07: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 125–134, Dubrovnik, Croatia, September 3–7, 2007. ACM.

[150] Jean-Claude Raoult and Jean Vuillemin. Operational and semantic equivalence between recursive programs. *Journal of the ACM (JACM)*, 27(4):772–796, October 1980. ACM.

[151] Matthias Rieger. *Effective Clone Detection Without Language Barriers*. PhD thesis, University of Bern, Switzerland, 2005.

[152] Chanchal K. Roy and James R. Cordy. A mutation/injection-based automatic framework for evaluating code clone detection tools. In *Mutation'09: Proceedings of the 4th International Workshop on Mutation Analysis, held in conjunction with ICST 2009 (the 2nd International Conference on Software Testing, Verification and Validation)*, pages 157–166, Denver, Colorado, USA, April 4, 2009. IEEE Computer Society.

[153] Filip Van Rysselberghe and Serge Demeyer. Evaluating clone detection techniques from a refactoring perspective. In *ASE'04: Proceedings of the 19th IEEE/ACM International Conference on Automated Software Engineering*, pages 336–339, Linz, Austria, September 20–25, 2004. IEEE Computer Society.

[154] Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel Quinlan, and Zhendong Su. Detecting code clones in binary executables. In *ISSTA'09: Proceedings of the 18th International Symposium on Software Testing and Analysis*, pages 117–128, Chicago, Illinois, USA, July 19–23, 2009. ACM.

[155] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: Local algorithms for document fingerprinting. In *SIGMOD'03: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 76–85, San Diego, California, USA, June 9–12, 2003. ACM.

[156] Markus Schordan and Daniel Quinlan. A source-to-source architecture for user-defined optimizations. In *JMLC'03: Proceedings of the Joint Modular Languages Conference*, volume 2789 of *Lecture Notes in Computer Science (LNCS)*, pages 214–223, Klagenfurt, Austria, August 24–27, 2003. Springer.

[157] David Schuler, Valentin Dallmeier, and Christian Lindig. A dynamic birthmark for Java. In *ASE'07: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 274–283, Atlanta, Georgia, USA, November 5–9, 2007. ACM.

[158] Andrew Schulman. Finding binary clones with opstrings and function digests. *Dr. Dobb's Journal*, 30(7–9):69–73, 56–61, 64–70, July–September 2005. Published by TechWeb, a division of United Business Media LLC.

[159] Jacob T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM (JACM)*, 27(4):701–717, October 1980. ACM.

[160] Robert C. Seacord, Daniel Plakosh, and Grace A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*. SEI Series in Software Engineering. Addison-Wesley, February 2003.

[161] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *ESEC/FSE'05: Proceedings of the 5th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 263–272, Lisbon, Portugal, September 5–9, 2005. ACM.

[162] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *SEC'01: Proceedings of the 10th USENIX Security Symposium*, pages 201–220, Washington, District of Columbia, USA, August 13–17, 2001. USENIX Association.

[163] Nija Shi and Ronald A. Olsson. Reverse engineering of design patterns from Java source code. In *ASE'06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 123–134, Tokyo, Japan, September 18–22, 2006. IEEE Computer Society.

[164] Ian Sommerville. *Software Engineering.* Addison Wesley, 6th edition, August 2000.

[165] Chad D. Sterling and Ronald A. Olsson. Automated bug isolation via program chipping. *Software: Practice and Experience (SP&E)*, 37(10):1061–1086, August 2007. John Wiley & Sons, Inc.

[166] Robert Tairas and Jeff Gray. Phoenix-based clone detection using suffix trees. In *ACM-SE 44: Proceedings of the 44th Annual Southeast Regional Conference*, pages 679–684, Melbourne, Florida, USA, March 10–12, 2006. ACM.

[167] Robert Tairas and Jeff Gray. An information retrieval process to aid in the analysis of code clones. *Empirical Software Engineering*, 14(1):33–56, February 2009. Springer.

[168] Robert Tairas, Jeff Gray, and Ira Baxter. Visualization of clone detection results. In *ETX'06: Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology eXchange*, pages 50–54, Portland, Oregon, USA, October 22–23, 2006. ACM.

[169] Robert Tairas, Shih hsi Liu, Frédéric Jouault, and Jeff Gray. CoCloRep: A DSL for code clones. In *ATEM'07: Proceedings of the 4th International Workshop on Software Language Engineering, held with MoDELS 2007 (the ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems)*, Nashville, Tennessee, USA, October 1, 2007. Springer.

[170] Michael Toomim, Andrew Begel, and Susan L. Graham. Managing duplicated code with linked editing. In *VLHCC'04: Proceedings of the 2004 IEEE Symposium on Visual Languages – Human Centric Computing*, pages 173–180, Rome, Italy, September 26–29, 2004. IEEE Computer Society.

[171] Esko Ukkonen. Approximate string-matching with q-grams and maximal matches. *Theoretical Computer Science, Selected Papers of the Combinatorial Pattern Matching School in Paris, France, July 1990*, 92(1):191–211, 1992. Elsevier.

[172] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with Java PathFinder. In *ISSTA'04: Proceedings of the ACM/SIGSOFT 2004 International Symposium on Software Testing and Analysis*, volume 29 of *SIGSOFT Software Engineering Notes*, pages 97–107, Boston, Massachusetts, USA, July 11–14, 2004. ACM.

[173] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS'00: Proceedings of the 7th Annual Symposium on Network and Distributed System Security*, pages 3–17, San Diego, California, USA, February 2–4, 2000. The Internet Society (ISOC).

[174] Vera Wahler, Dietmar Seipel, Jürgen Wolff von Gudenberg, and Gregor Fischer. Clone detection in source code by frequent itemset techniques. In *SCAM'04: Proceedings of the 4th IEEE International Workshop on Source Code Analysis and Manipulation*, pages 128–135, Chicago, Illinois, USA, September 15–16, 2004. IEEE Computer Society.

[175] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering (TSE)*, 10(4):352–357, July 1984. IEEE Computer Society.

[176] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *TACAS'05: Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, held as part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2005*, volume 3440 of *Lecture Notes in Computer Science (LNCS)*, pages 365–381, Edinburgh, UK, March 25 – April 2, 2005. Springer.

[177] Yichen Xie and Alexander Aiken. Context- and path-sensitive memory leak detection. In *ESEC/FSE'05: Proceedings of the 5th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 115–125, Lisbon, Portugal, September 5–9, 2005. ACM.

[178] Yichen Xie and Alexander Aiken. Scalable error detection using boolean satisfiability. In *POPL'05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 351–363, Long Beach, California, USA, January 12–14, 2005. ACM.

[179] Yichen Xie and Dawson R. Engler. Using redundancies to find errors. In *FSE'02: Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 51–60, Charleston, South Carolina, USA, November 18–22, 2002. ACM.

[180] Bin Xin and Xiangyu Zhang. Efficient online detection of dynamic control dependence. In *ISSTA'07: Proceedings of the 16th International Symposium on Software Testing and Analysis*, pages 185–195, London, UK, July 9–12, 2007. ACM.

[181] Bin Xin and Xiangyu Zhang. Memory slicing. In *ISSTA'09: Proceedings of the 18th International Symposium on Software Testing and Analysis*, pages 165–176, Chicago, Illinois, USA, July 19–23, 2009. ACM.

[182] Xifeng Yan, Jiawei Han, and Ramin Afshar. CloSpan: Mining closed sequential patterns in large databases. In *SDM'03: Proceedings of the 3rd SIAM International Conference on Data Mining*, pages 166–177, San Francisco, California, USA, May 1–3, 2003. SIAM.

[183] Rui Yang, Panos Kalnis, and Anthony K. H. Tung. Similarity evaluation on tree-structured data. In *SIGMOD'05: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 754–765, Baltimore, Maryland, USA, June 14–16, 2005. ACM.

[184] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *CCS'07: Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 116–127, Alexandria, Virginia, USA, October 28–31, 2007. ACM.

[185] Norihiro Yoshida, Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. On refactoring support based on code clone dependency relation. In *METRICS'05: Proceedings of the 11th IEEE International Software Metrics Symposium*, page 16, Como, Italy, September 19–22, 2005. IEEE Computer Society.

[186] Liguo Yu and Srini Ramaswamy. Improving modularity by refactoring code clones: A feasibility study on linux. *SIGSOFT Software Engineering Notes for the 1st International Global Requirements Engineering Workshop (GREW'07)*, 33(2):1–5, March 2008. ACM.

[187] Vladimir A. Zakharov. To the functional equivalence of turing machines. In *FCT'87: Proceedings of the 1987 International Conference on Fundamentals of Computation*

Theory, volume 278 of *Lecture Notes in Computer Science (LNCS)*, pages 488–491, Kazan, USSR, June 22–26, 1987. Springer.

[188] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering (TSE)*, 28(2):183–200, February 2002. IEEE Computer Society.

[189] Kaizhong Zhang and Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing (SICOMP)*, 18(6):1245–1262, December 1989. Society for Industrial and Applied Mathematics.

[190] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Pruning dynamic slices with confidence. In *PLDI'06: Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, pages 169–180, Ottawa, Ontario, Canada, June 11–14, 2006. ACM.

[191] Xiaoming Zhou, Xingming Sun, Guang Sun, and Ying Yang. A combined static and dynamic software birthmark based on component dependence graph. In *IIH-MSP-2008: Proceedings of the 4th International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, volume 2, pages 1416–1421, Harbin, China, August 15–17, 2008. IEEE.

[192] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *ICSE'04: Proceedings of the 26th International Conference on Software Engineering*, pages 563–572, Edinburgh, Scotland, UK, May 23–28, 2004. IEEE Computer Society.

[193] Richard Zippel. An explicit separation of relativised random polynomial time and relativised deterministic polynomial time. *Information Processing Letters*, 33(4):207–212, December 1989. Elsevier.