CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code

Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou, *Member*, *IEEE*

Abstract—Recent studies have shown that large software suites contain significant amounts of replicated code. It is assumed that some of this replication is due to copy-and-paste activity and that a significant proportion of bugs in operating systems are due to copy-paste errors. Existing static code analyzers are either not scalable to large software suites or do not perform robustly where replicated code is modified with insertions and deletions. Furthermore, the existing tools do not detect copy-paste related bugs. In this paper, we propose a tool, CP-Miner, that uses data mining techniques to efficiently identify copy-pasted code in large software suites and detects copy-paste bugs. Specifically, it takes less than 20 minutes for CP-Miner to identify 190,000 copy-pasted segments in Linux and 150,000 in FreeBSD. Moreover, CP-Miner has detected many *new* bugs in popular operating systems, 49 in Linux and 31 in FreeBSD, most of which have since been confirmed by the corresponding developers and have been rectified in the following releases. In addition, we have found some interesting characteristics of copy-paste in operating system code. Specifically, we analyze the distribution of copy-pasted code by size (number lines of code), granularity (basic blocks and functions), and modification within copy-pasted code. We also analyze copy-paste across different modules and various software versions.

Index Terms—Software analysis, code reuse, code duplication, debugging aids, data mining.

1 Introduction

1.1 Motivation

Recent studies [2], [3], [4] have shown that a large portion of code appears to be duplicated in software. For example, Kapser and Godfrey [4], using a code clone detection tool called CCFinder [5], found that 12 percent of the Linux file system code (279K lines) was involved in code cloning activity. Baker [2] found that, in the complete source of the X Window system (714K lines), 19 percent of the code was identified as duplicates. Duplicated code is likely to result from copy-paste activity because it can significantly reduce programming effort and time by reusing a piece of code rather than rewriting similar code from scratch. This practice is common, especially in device drivers of operating systems where the algorithms are similar. In addition, performance enhancement, coding style, and accidents are also the reasons why large systems contain a large portion of duplicated code [6].

Using abstractions such as functions and macros to improve software maintenance might remove code duplication; however, much duplicated code will likely remain for two main reasons. First, some changes are usually necessary in different copies and copy-paste is much easier and faster than abstraction. Second, functions may impose higher overhead in execution.

While one can imagine augmenting software development tools and editors with copy-paste tracking, this support does not currently exist. Without such tracking, it is difficult to tell whether a duplicated code segment is really the result of copy-paste activity. For simplicity, in this paper, we use the term "**copy-pasted code**" to refer to duplicated code in general and the term "copy-paste" refers to copy-and-paste activity.

Copy-pasted code is prone to introducing errors. For example, Chou et al. [7] found that, in a single source file under the Linux **drivers/i2o** directory, 34 out of 35 errors were caused by copy-paste. One of the errors was copied in 10 places and another in 24. They also showed that many operating system errors are not independent because programmers are ignorant of system restrictions in copy-pasted code. In our study, we have detected 49 copy-paste related bugs in the latest version of Linux and 31 in FreeBSD. Most of these bugs were previously unreported.

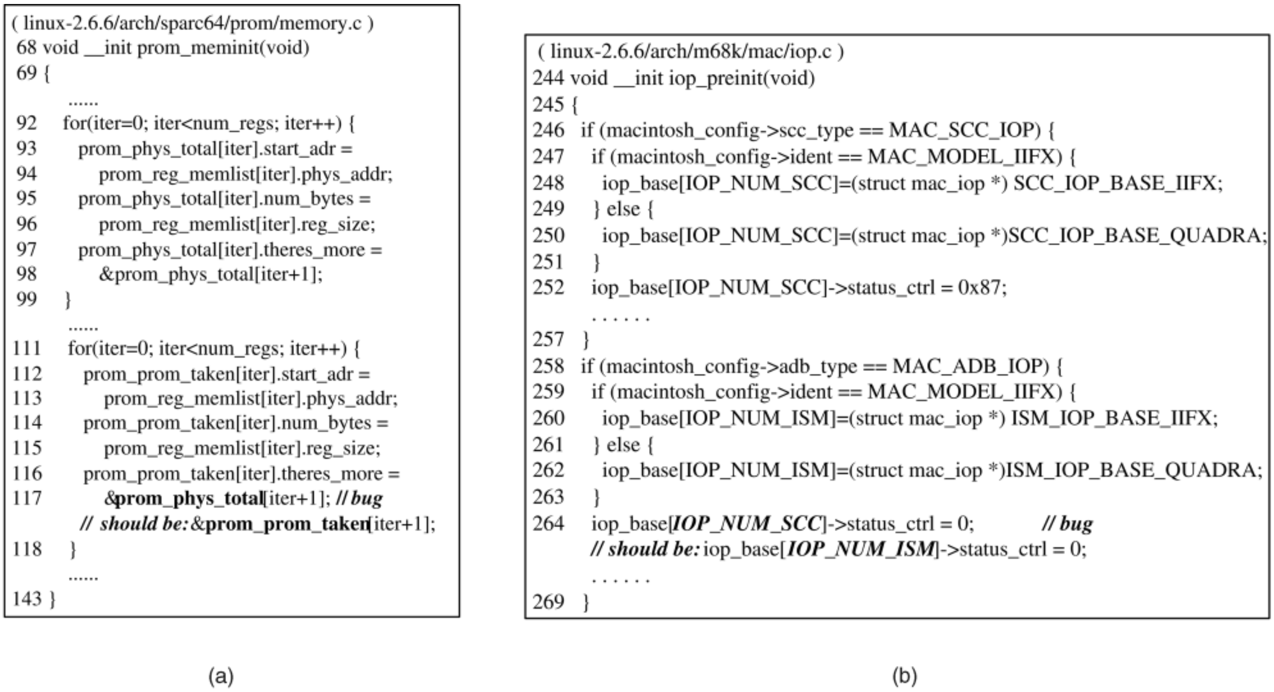
One of the major reasons why copy-paste introduces bugs is that programmers forget to modify identifiers (variables, functions, types, etc.) consistently throughout the pasted code. This mistake will be detected by a compiler if the identifier is undefined or has the wrong type. However, these errors often slip through compile-time checks and become hidden bugs that are very hard to detect.

Fig. 1a shows an example of a *new* bug detected by CP-Miner in the latest version of Linux (2.6.6). We reported this bug to the Linux kernel community and it has been confirmed and fixed by the kernel developers [8]. In this example, the loop in lines 111-118 was copied from lines 92-99. In the new copy-pasted segment (lines 111-118), the variable *prom_phys_total* is replaced with *prom_prom_taken* in most of the cases except the one in line 117 (shown in bold font). As a result, the pointer *prom_prom_taken[iter].theres_more* incorrectly points to the element of *prom_phys_total* instead of *prom_prom_taken*. This bug is a semantic error and, therefore, it cannot be easily detected by memory-related bug detection tools, including static checkers [9], [10], [11], [12] or dynamic tools such as Purify [13], Valgrind [14], and CCured [15]. Besides this bug, CP-Miner has also detected many other

- *The authors are with the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801.*
E-mail: {zli4, shanlu, myagmar, yyzhou}@uiuc.edu.

Manuscript received 2 Aug. 2005; revised 10 Jan. 2006; accepted 13 Jan. 2006; published online DD Mmmm, YYYY.
Recommended for acceptance by M. Harman.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0210-0805.

0098-5589/06/$20.00 © 2006 IEEE Published by the IEEE Computer Society



(a)

```
( linux-2.6.6/arch/sparc64/prom/memory.c )
68 void __init prom_meminit(void)
69 {
      ......
92    for(iter=0; iter<num_regs; iter++) {
93      prom_phys_total[iter].start_adr =
94          prom_reg_memlist[iter].phys_addr;
95      prom_phys_total[iter].num_bytes =
96          prom_reg_memlist[iter].reg_size;
97      prom_phys_total[iter].theres_more =
98          &prom_phys_total[iter+1];
99    }
      ......
111   for(iter=0; iter<num_regs; iter++) {
112     prom_prom_taken[iter].start_adr =
113         prom_reg_memlist[iter].phys_addr;
114     prom_prom_taken[iter].num_bytes =
115         prom_reg_memlist[iter].reg_size;
116     prom_prom_taken[iter].theres_more =
117         &prom_phys_total[iter+1]; // bug
        // should be:&prom_prom_taken[iter+1];
118   }
      ......
143 }
```

(b)

```
( linux-2.6.6/arch/m68k/mac/iop.c )
244 void __init iop_preinit(void)
245 {
246  if (macintosh_config->scc_type == MAC_SCC_IOP) {
247    if (macintosh_config->ident == MAC_MODEL_IIFX) {
248      iop_base[IOP_NUM_SCC]=(struct mac_iop *) SCC_IOP_BASE_IIFX;
249    } else {
250      iop_base[IOP_NUM_SCC]=(struct mac_iop *)SCC_IOP_BASE_QUADRA;
251    }
252    iop_base[IOP_NUM_SCC]->status_ctrl = 0x87;
       . . . . . .
257  }
258  if (macintosh_config->adb_type == MAC_ADB_IOP) {
259    if (macintosh_config->ident == MAC_MODEL_IIFX) {
260      iop_base[IOP_NUM_ISM]=(struct mac_iop *) ISM_IOP_BASE_IIFX;
261    } else {
262      iop_base[IOP_NUM_ISM]=(struct mac_iop *)ISM_IOP_BASE_QUADRA;
263    }
264    iop_base[IOP_NUM_SCC]->status_ctrl = 0;          // bug
       // should be: iop_base[IOP_NUM_ISM]->status_ctrl = 0;
       . . . . . .
269  }
```

Fig. 1. Copy-paste related bugs in Linux 2.6.6 detected by CP-Miner. These bugs have been confirmed and fixed by Linux kernel developers. (a) Detected in file /arch/sparc64/prom/memory.c. A similar bug is also detected in file/arch/sparc/prom/memory.c. (b) Detected in file /arch/m68k/mac/iop.c.

similar bugs caused by copy-paste in Linux, FreeBSD, PostgreSQL, and Web Apache.

Another copy-paste related bug detected by CP-Miner is shown in Fig. 1b. In this example, the segment in lines 258-269 was copied from lines 246-257. Each segment initializes different IOPs (I/O processors) specified by constants *IOP_NUM_SCC* (= 0) and *IOP_NUM_ISM* (= 1), respectively. However, the identifier *IOP_NUM_SCC* in line 264 is not changed to *IOP_NUM_ISM* accordingly and it results in a wrong initial state of IOPs. This bug would incorrectly overwrite the value (0x87) of *iop_base[IOP_NUM_SCC]->status_ctrl* by 0. This cannot be detected by existing bug detection tools because it is not a simple buffer overflow bug (since *IOP_NUM_SCC* equals 0), incorrect pointer manipulation, or free memory access. If known by malicious users who plan a security attack, this bug may cause the server to crash.

It is a challenging task to efficiently extract copy-pasted code in large software suites such as an operating system. Even though several previous studies [16], [17] have addressed the related problem of detecting plagiarism, they are not suitable for detecting copy-pasted code. Those tools, such as the commonly used JPlag [18], were designed to measure the degree of similarity between a pair of programs in order to detect plagiarism. If these tools were to be used to detect copy-pasted code in a single program *without any modification*, they would need to compare all possible pairs of code fragments. For a program with n statements, a total of $O(n^3)$ pairwise comparisons[1] would need to be performed. This complexity is certainly impractical for software with millions of lines of code, such as Linux and FreeBSD. Of course, it is possible to modify these tools to identify copy-pasted code in single software, but the modification is not trivial and straightforward. For example, a new dynamic programming algorithm may be integrated into the original detection algorithm, which would require significant effort.

So far, only a few tools have been proposed to identify copy-pasted code in a single program. Examples of such tools include Moss [19], [20], Dup [2], CCFinder [5], and others [6], [21]. Most of these tools suffer from some or all of the following limitations:

1. *Efficiency.* Most existing tools are not scalable to large software suites such as operating system code because they consume a large amount of memory and take a long time to analyze millions of lines of code.
2. *Tolerance to modifications.* Most tools cannot deal with modifications in copy-pasted code. Some tools [3], [22] can only detect copy-pasted segments that are exactly identical. Such modifications are very common in standard practice. Our experiments with CP-Miner show that about one third of copy-pasted segments contain insertion or modification of one to two statements.
3. *Bug detection.* Although some existing tools report copy-pasted code, they cannot detect copy-paste related bugs.

1.2 Our Contributions

In this paper, we present CP-Miner, a tool that uses data mining techniques to *efficiently* identify copy-pasted code in large software suites including operating system code and also detects copy-paste related bugs. It requires no

[1] Considering comparison between the pair of code fragments with k statements, there are $(n-k+1)$ different fragments. So, there are $\binom{n-k+1}{2} = O(n^2)$ possible pair comparisons. Since k can be $1, 2, \ldots, \frac{n}{2}$, the total number of pairwise comparisons is $O(n^3)$.

modification or annotation to the source code of the software being analyzed. Our work makes three main contributions:

1. **A scalable copy-paste detection tool for large software suites.** CP-Miner can efficiently find copy-pasted code in large software suites including operating system code. Our experimental results show that it takes less than 20 minutes for CP-Miner to detect 200,000 and 150,000 unique copy-pasted segments that account for about 22 percent and 20 percent of the source code in Linux and FreeBSD (each with more than 3 million lines of code), respectively. Additionally, it takes less than one minute to detect copy-pasted segments in Apache web server and PostgreSQL, accounting for about 18 percent and 22 percent of the total source code, respectively.

   Compared to CCFinder [5], CP-Miner is able to find 17-52 percent more copy-pasted segments in the four test applications because CP-Miner can tolerate statement insertions and modifications.
2. **Detection of bugs associated with copy-paste.** CP-Miner can detect copy-paste related bugs such as the bugs shown in Fig. 1, most of which are hard to detect with existing static or dynamic bug detection tools. Specifically, CP-Miner has detected 49 new bugs in the latest version of Linux, 31 in FreeBSD, 5 in Web Apache, and 2 in PostgreSQL. These bugs had not been reported before.

   We have reported these bugs to the corresponding developers. So far, most of these bugs have been confirmed and fixed by Linux and FreeBSD developers and have been rectified in the following releases.
3. **Statistical study of copy-pasted code distribution in operating system code.** Few earlier studies have been conducted on the characteristics of copy-paste in large software suites. Our work found some interesting characteristics of copy-pasted code in Linux and FreeBSD. Our results indicate that:

   a. copy-pasted segments are usually not too large, most with 5-16 statements;
   b. although more than 50 percent of copy-pasted segments have only two copies, a few (6.3-6.7 percent) copy-pasted segments are copied more than eight times;
   c. there are a significant number (11.3-13.5 percent) of copy-pasted segments at function granularity (copy-paste of an entire function);
   d. most (65-67 percent) copy-pasted segments require renaming at least one identifier and 23-27 percent of copy-pasted segments have inserted, modified (excluding renaming), or deleted one statement;
   e. different OS modules have very different percentages of copy-pasted code: *drivers*, *arch*, and *crypt* have higher percentage of copy-paste than other modules in Linux; and
   f. as the operating system code evolves, the amount of copy-paste also increases, but the coverage percentage of copy-pasted code remains relatively stable over the recent versions of Linux and FreeBSD.

2 Related Work and Background

2.1 Detection of Copy-Pasted Code

Since copy-pasted code segments are usually similar to the original ones, detection of copy-pasted code involves detecting code segments that are identical or similar.

Previous techniques for copy-paste detection can be roughly classified into three categories: 1) *string-based*, in which the program is divided into strings (typically lines) and these strings are compared against each other to find sequences of duplicated strings [2], [23]; 2) *parse-tree-based*, in which pattern matching is performed on the parse-tree of the code to search for similar subtrees [6], [24], [25]; 3) *token-based*, in which the program is divided into a stream of tokens and duplicate token sequences are identified [5], [18].

Dup, proposed by Baker [2], finds all pairs of matching *parameterized* code fragments. A code fragment matches another if both fragments are contiguous sequences of source lines with some consistent identifier mapping scheme. Because this approach is line-based, it is sensitive to lexical aspects such as the presence or absence of new lines. In addition, it does not find noncontiguous copy-pastes. CP-Miner does not have these shortcomings.

Johnson [23] proposed using a fingerprinting algorithm on a substring of the source code. In this algorithm, signatures calculated per line are compared in order to identify matched substrings. As with line-based techniques, this approach is sensitive to minor modifications made in copy-pasted code.

Baxter et al. [6] proposed a tool that transforms source code into abstract-syntax trees (AST) and detects copy-paste by finding identical subtrees. Similarly to other tools, it cannot perform robustly when modifications are made in copy-pasted segments.

Komondoor and Horwitz [24] proposed using a program dependence graph (PDG) and program slicing to find isomorphic subgraphs and code duplication. Although this approach is successful at identifying copies with reordered statements, its running time is very long. For example, it takes 1.5 hours to analyze only 11,540 lines of source code of *bison*, much slower than CP-Miner. Another slow PDG-based approach is found in [26].

Kontogiannis et al. [25] built an abstract pattern matching tool to identify probable matches using Markov models. This approach does not find copy-pasted code. Instead, it only measures similarity between two programs.

Mayrand et al. [27] used an Intermediate Representation Language to characterize each function in the source code and detect copy-pasted function bodies that have similar metric values. This tool does not detect copy-paste at other granularity such as segment-based copy-paste, which occurs more frequently than function-based copy-paste, as shown in our results.

Some copy-paste detection techniques are too coarse-grained to be useful for our purpose. *JPlag* [18], *Moss* [20], and *sif* [28] are tools to find similar programs among a given set. They have been commonly used to detect plagiarism. Most of them are unscalable for detecting copy-pasted code in a single large program.

Some graphical tools were proposed to understand code similarities in different programs (or in the same program) visually. *Dotplots*, proposed by Church and Helfman [29], can visualize similar code by tokenizing the code into lines and placing a dot in coordinates (i, j) on a 2D graph, if the $i\mathrm{th}$ input token matches $j\mathrm{th}$ input token. Similarly, *Duploc* proposed by Ducasse et al. [3] provides a scatter plot visualization of copy-pastes (detected by string matching of lines) and also textual reports that summarize all discovered sequences. Both *Dotplots* and *Duploc* only support line granularity. In addition, they can only detect identical duplicates and do not tolerate renaming, insertions, and deletions.

Our tool, CP-Miner, is token-based. This approach has advantages over the other two. First, a string-based approach does not exploit any lexical information, so it cannot deal with simple modifications such as identifier renaming. Second, using parse trees can introduce false positives because two segments with identical syntax trees may not be copy-pasted. Therefore, it also has to compare the subtrees by matching tokens. This method is not scalable because the complexity of comparing sequences of trees is $O(N^4)$. Although the scalability issue of parse-tree-based method can be tackled by hashing subtrees [6], it cannot perform robustly when the duplicated code contains modifications. CP-Miner also leverages some source code level and syntax information to make the analysis more accurate. This will be described in detail in Section 3.1.

Most previous copy-paste detection tools do not sufficiently address the limitations described in Section 1. Most of them consume too much time or memory to be scalable to large applications or do not tolerate modifications made in copy-pasted code. In contrast, CP-Miner can address both challenges by using frequent subsequence mining techniques (described in Section 3.1).

2.2 Detecting Software Bugs

Many tools have been proposed for detecting software bugs. One approach is dynamic checking that detects bugs during execution. Examples of dynamic tools include Purify [13], Valgrind [14], DIDUCE [30], Eraser [31], and CCured [15]. Dynamic tools provide more accurate results but may introduce significant overheads during execution. Moreover, they can only find bugs on the execution paths. Most dynamic tools cannot detect bugs in operating systems.

Another approach is to perform checks statically. Examples of this approach include explicit model checking [12], [32], [33] and program analysis [10], [11], [34]. Most static tools require significant work from programmers to write specifications or annotate source code. But, the advantage of static tools is that they add no overhead during execution and they can find bugs that may not occur in the common execution paths. A few tools do not require annotations, but they focus on detecting other types of bugs instead of copy-paste related bugs.

Our tool, CP-Miner, is a static tool that can detect copy-paste related bugs, *without requiring any annotations by programmers*. CP-Miner complements other bug detection tools because it is based on a different observation: finding bugs caused by copy-paste. Some copy-paste related bugs can be found by previous tools if they lead to buffer overflow or some obvious memory corruption, but many of them, especially semantic ones, cannot be found by previous tools.

Our work is motivated by and related to Chou et al.'s empirical analysis of operating systems errors [7]. Their study gave an overall error distribution and evolution analysis in operating systems and found that copy-paste is one of the major causes of bugs. Our work presents a tool for detecting copy-pasted code and related bugs in large-scale software including operating system code. Many of these bugs, such as the two shown in Fig. 1, cannot be detected by their tools.

2.3 Frequent Subsequence Mining

CP-Miner is based on *frequent subsequence mining* (also called frequent sequence mining), which is an association analysis technique to discover frequent subsequences in a sequence database [35]. Frequent subsequence mining is an active research topic in data mining [36]. It has broad applications, including mining motifs in DNA sequences, analysis of customer shopping behavior, etc.

A *sequence* is an abstraction over elements which appear in a consecutive order. A *subsequence* is any subset of these elements which appear in the same order, that is, the elements of a subsequence may be interleaved in the original sequence. The notion of a *frequent subsequence* is one which is interleaved in a number of sequences. The sequence in which a given subsequence appears is called a *supporting sequence*. The number of the sequences in which a subsequence so appears is known as its *support*. Qualification of a subsequence as a frequent subsequence is therefore determined by setting a threshold, called *min_support*, such that all subsequences whose support is greater than or equal to this value are considered frequent.

For example, a sequence database D has five sequences:

$$D = \{abced, abecf, agbch, abijc, aklc\}.$$

The number of occurrences of subsequence abc is 4 and sequence $agbch$ is one of abc's supporting sequences. If $min\_support$ is specified as 4, the frequent subsequences are $\{a{:}\,5, b{:}\,4, c{:}\,5, ab{:}\,4, ac{:}\,5, bc{:}\,4, abc{:}\,4\}$, where the numbers are the supports of the subsequences.

CP-Miner uses a recently proposed frequent subsequence mining algorithm called *CloSpan* (Closed Sequential Pattern Mining) [36], which outperforms most previous algorithms. *Closed subsequence* is the subsequence whose support is different from the support of its supersequences. In other words, nonclosed subsequences can be inferred from their supersequences with the same support. CloSpan consists of two main stages: 1) using a depth-first search procedure to generate a candidate set of frequent subsequences that includes all the closed frequent subsequences and 2) pruning the nonclosed subsequences from the candidate set. The computational complexity of CloSpan is $O(n^2)$ if the maximum length of frequent sequences is constrained by a constant.

CloSpan produces only closed frequent subsequences rather than all frequent subsequences since any nonclosed subsequence can be inferred from its supersequence. In the example above, the frequent subsequences are $\{a{:}\,5, b{:}\,4, c{:}\,5, ab{:}\,4, ac{:}\,5, bc{:}\,4, abc{:}\,4\}$ if the $min\_support$ is 4, but we only need to produce the closed subsequences $\{ac{:}\,5, abc{:}\,4\}$. This feature significantly reduces the number of frequent subsequences generated, especially for long frequent subsequences.

There are two main ideas to improve the mining efficiency in CloSpan. The first idea is based on an obvious observation that if a sequence is frequent, then all of its subsequences are frequent. For example, if a sequence abc is frequent, all of its subsequences $\{a, b, c, ab, ac, bc\}$ are frequent. CloSpan recursively produces a longer frequent subsequence by concatenating every frequent item with a shorter frequent subsequence that has already been obtained in the previous iterations.

To better explain this idea, let us consider an example. Let L_n denote the set of frequent subsequences with length n. In order to get L_n, we can join the sets L_{n-1} and L_1. For example, suppose we have already computed L_1 and L_2 as shown below. In order to compute L_3, we can first compute L'_3 by concatenating a subsequence from L_2 and an item from L_1:

$$L_1 = \{a, b, c\},$$
$$L_2 = \{ab, ac, bc\},$$
$$L'_3 = L_2 \times L_1 = \{aba, abb, abc, aca, acb, acc, bca, bcb, bcc\}.$$

For greater efficiency, CloSpan does not join the sequences in set L_2 with all the items in L_1. Instead, each sequence in L_2 is concatenated only with the frequent items in its suffix database. *Suffix database* for a subsequence s is composed of all suffixes of the sequences containing s in the original database. In our example, for the frequent sequence ab in L_2, its suffix database is $D_{ab} = \{ced, ecf, ch, ijc\}$ and only c is a frequent item in D_{ab} (its support is equal to $min\_support = 4$), so ab is only concatenated with c and then we get a longer sequence abc that belongs to L'_3.

The second idea is used to efficiently evaluate whether a concatenated subsequence is frequent or not. It tries to avoid searching through the whole database. Instead, it only checks suffix databases that can be created by scanning the whole database *once* at the beginning and then updated when frequent subsequences are generated. In the above example, for each sequence s in L'_3, CloSpan checks whether it is frequent or not by searching the suffix database D_s. If the number of its occurrences is greater than $min\_support$, s is added into L_3, which is the set of frequent subsequences of length 3. CloSpan continues computing L_4 from L_3, L_5 from L_4, and so on until no more subsequences can be added into the set of frequent subsequences. In the example above, the algorithm stops at $L_3 = \{abc\}$ since there is no frequent items in suffix database $D_{abc} = \{ed, f, h\}$, where D_{abc} can be easily obtained from suffix database $D_{ab} = \{ced, ecf, ch, ijc\}$ without scanning the whole database D again.

Recently, we have used CloSpan to detect block correlations in storage systems [37]. We do not discuss the CloSpan algorithm in more detail as it can be found in [36].

3 CP-MINER

CP-Miner has two major functions: detecting copy-pasted code segments and finding copy-paste related bugs. It requires no modification in the source code of the software being analyzed. The following two subsections describe the design of each function.

3.1 Identifying Copy-Pasted Code

To detect copy-pasted code, CP-Miner first converts the problem into a frequent subsequence mining problem. It then uses an enhanced algorithm of CloSpan to find *basic* copy-pasted segments. Finally, it prunes *false positives* that are unlikely to be real copy-pasted code and then composes larger copy-pasted segments. For convenience, we refer to a group of code segments that are similar to each other as a *copy-paste group*.

CP-Miner can detect copy-pasted segments *efficiently* because it uses frequent subsequence mining techniques that can avoid many unnecessary or redundant comparisons. To map our problem to a frequent subsequence mining problem, CP-Miner first maps a statement to a number, with similar statements being mapped to the same number (described in Section 3.1.1). Then, a basic block (i.e., a straightline piece of code without any jumps or jump targets in the middle) becomes a sequence of numbers. As a result, a program is mapped into a database of many sequences. By mining the database using CloSpan, we can find frequent subsequences that occur at least twice in the sequence database. These frequent subsequences are exactly copy-pasted segments in the original program. By applying some pruning techniques, we can find basic copy-pasted segments, which then can be combined with neighboring ones to compose larger copy-pasted segments.

CP-Miner is capable of handling modifications in copy-pasted segments for two reasons. First, similar statements are mapped into the same value. This is achieved by mapping all identifiers (variables, functions, types, etc.) of the same type into the same value, regardless of their actual names. This relaxation tolerates identifier renaming in copy-pasted segments. Even though false positives may be introduced during this process, they are addressed later through various pruning techniques, such as identifier mapping (described in Section 3.1.4). Second, a frequent subsequence can be interleaved in its supporting sequences. Since the mining algorithm allows arbitrary interleaving gaps in the sequences, we enhance the basic mining algorithm, CloSpan, to support gap constraints in frequent subsequences. This enhancement allows CP-Miner to tolerate one to two statement insertions, deletions, or modifications in copy-pasted code, ignoring an arbitrarily long different code segment that is unlikely to be copy-pasted code. Insertions and deletions are symmetric because a statement deletion in one copy can also be seen as an insertion in the other copy. Modification is a special case of insertion. Basically, the modified statement can be treated as if both segments have an inserted statement.

The main steps in the process of identifying copy-pasted segments include:

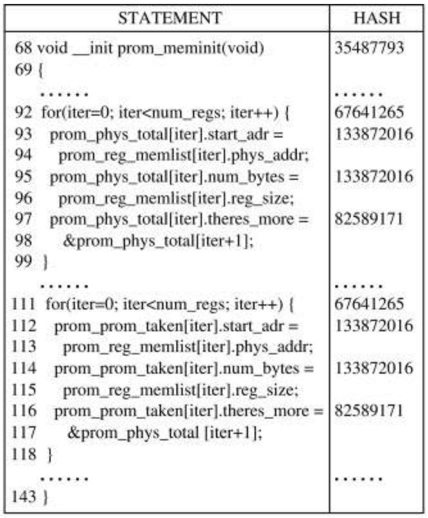
1. *Parsing source code.* Parse the given source code and build a sequence database (a collection of sequences). In addition, information regarding basic blocks and block nesting levels is also passed to the mining algorithm.
2. *Mining for basic copy-pasted segments.* The enhanced frequent subsequence mining algorithm is applied to the sequence database to find basic copy-pasted segments.
3. *Pruning false positives.* Some code segments appear to be duplicated due to their similarity in syntax structure, but are actually not duplicated code. These code segments are false positives. Various techniques are used to weed them out.
4. *Composing larger copy-pasted segments.* Larger copy-pasted segments are identified by combining consecutive smaller ones. The combined copy-pasted segments are fed back to Step 3 to prune false positives. This is necessary because the composite segment may not be copy-pasted, even though its components are.

Like other copy-paste detection tools, CP-Miner can only detect copy-pasted segments, but cannot tell which segment is the original and which is copy-pasted from the original. Fortunately, this is not a significant limitation because, in most cases, it is enough for programmers to know which segments are similar to each other. Moreover, our bug detection method described in Section 3.2 does not rely on such differentiation. Additionally, if programmers really need the differentiation, navigating through RCS versions could help in figuring out which segment is the original copy.

3.1.1 Parsing Source Code

The main purpose of parsing source code is to build a sequence database (a collection of sequences) in order to convert the copy-paste detection problem to a frequent subsequence mining problem. Comments are not considered normal statements in CP-Miner and are thereby filtered by our parser. Comments are valuable information to indicate *genuine* copying. For example, plagiarism detection tools can use them as clues in code where identifiers have been renamed. CP-Miner can be enhanced by exploiting such information to increase the confidence of copy-paste detection. The current prototype of the CP-Miner parser only works for programs written in C or C++, but it is not difficult to adapt to other programming languages by using the corresponding parsers with a little modification.

A statement is mapped to a number by first tokenizing its components, such as variables, operators, constants, functions, keywords, etc. To tolerate identifier renaming in copy-pasted segments, identifiers of the same type (such as variable, function, type, etc., not data type) are mapped into the same token. Notice that the variables with different data type are mapped into the same token because the data types in copy-pasted code segments can be changed (e.g., the variables can be changed from *int* to *float* and other data types); similarly, all function names are mapped into the same token. Constants are handled in the same way as identifiers: Constants are mapped into the same token. However, operators and keywords are handled differently, with each one mapped into a unique token. After all the components of a statement are tokenized, a hash value digest is computed using the "hashpjw" hash function proposed by Weinberger (see [38]), chosen for its low collision rate. Fig. 2 shows the hash value for each statement in the example shown in Fig. 1a of Section 1. As shown in this figure, the statement in lines 93-94 and the statement in lines 112-113 have the same hash values.



STATEMENT	HASH
68 void __init prom_meminit(void)	35487793
69 {	
......	......
92 for(iter=0; iter<num_regs; iter++) {	67641265
93 prom_phys_total[iter].start_adr =	133872016
94 prom_reg_memlist[iter].phys_addr;	
95 prom_phys_total[iter].num_bytes =	133872016
96 prom_reg_memlist[iter].reg_size;	
97 prom_phys_total[iter].theres_more =	82589171
98 &prom_phys_total[iter+1];	
99 }	
......	......
111 for(iter=0; iter<num_regs; iter++) {	67641265
112 prom_prom_taken[iter].start_adr =	133872016
113 prom_reg_memlist[iter].phys_addr;	
114 prom_prom_taken[iter].num_bytes =	133872016
115 prom_reg_memlist[iter].reg_size;	
116 prom_prom_taken[iter].theres_more =	82589171
117 &prom_phys_total [iter+1];	
118 }	
......	......
143 }	

Fig. 2. An example of hashing statements.

After each statement is mapped, the program becomes a long sequence of numbers. Unfortunately, the frequent subsequence mining algorithms need a collection of sequences (a sequence database), as described in Section 2.3, so we need a way to cut this long sequence into many short ones. One simple method is to use a fixed cutting window size (e.g., every 20 statements) to break the long sequence into many short ones. This method has two disadvantages. First, some frequent subsequences across two or more windows may be lost. Second, it is not easy to decide the window size: If it is too long, the mining algorithm would be very slow; if too short, too much information may be lost on the boundary of two consecutive windows.

Instead, CP-Miner uses a more elegant method to perform the cutting. It takes advantage of some simple syntax information and uses a basic programming block as a unit to break the long sequence into short ones. The idea for this cutting method is that a copy-pasted segment is usually either a part of a basic block or consists of multiple basic blocks. In addition, basic blocks are usually not too long to cause performance problems in CloSpan. Although it also exploits syntax information, it is different from the parse-tree-based approach (described in Section 2.1) in that CP-Miner does not compare the structure of abstract syntax trees and, thus, avoids the high complexity in the parse-tree-based approach.

By using a basic block as the cutting unit, CP-Miner can first find basic copy-pasted segments and then compose larger ones from smaller ones. Since different basic blocks have a different number of statements, their corresponding sequences also have different lengths. But, this is not a problem for CloSpan because it can deal with sequences of different sizes. The example shown in Fig. 2 is converted into the following collection of sequences:

(35487793)
......
(67641265)
(133872016, 133872016, 82589171)
......
(67641265)
(133872016, 133872016, 82589171)
......

Besides a collection of sequences, the parser also passes to the mining algorithm the source code information of each sequence. Such information includes 1) the nesting level of each basic block, which is later used to guide the composition of larger copy-pasted segments from smaller ones and 2) the file name and line number, which is used to locate the copy-pasted code corresponding to a frequent subsequence identified by the mining algorithm. We also keep the source code identifiers associated with each hashed statement to later apply identifier mapping between code segment pairs without parsing the source code again.

3.1.2 Mining for Basic Copy-Pasted Segments

After CP-Miner parses the source code of a given program, it generates a sequence database with each sequence representing a basic block. In the next step, it applies the frequent subsequence mining algorithm, CloSpan, on this database to find frequent subsequences with support value of at least 2, which corresponds to code segments that have appeared in the program at least twice. In the example shown in Fig. 2, CP-Miner would find (133872016, 133872016, 82589171) as a frequent subsequence because it occurs twice in the sequence database. Therefore, the corresponding code segments in lines 111-118 and lines 92-99 are basic copy-pasted segments.

Unfortunately, the mining process is not as straightforward as it would seem. The main reason is that the original CloSpan algorithm was not designed exactly for our purpose nor were other frequent subsequence mining algorithms. Most existing algorithms, including CloSpan, have the following two limitations that we had to overcome in CloSpan to make it applicable for copy-paste detection:

1. **Adding gap constraints in frequent subsequences.** In most existing frequent subsequence mining algorithms, frequent subsequences are not necessarily contiguous in their supporting sequences. For example, sequence $abdec$ provides one support for subsequence abc, even though abc does not appear contiguously in $abdec$. This property in frequent sequence mining provides CP-Miner with the capability of finding some true copy-paste with slight modification and challenge to tolerating false positives introduced by too much difference in a code segment.

   To address this problem, we modified CloSpan to add a gap constraint in frequent subsequences. CP-Miner only mines for frequent subsequences with a maximum gap not larger than a given threshold, called max_gap. If the maximum gap of a subsequence in a sequence is larger than max_gap, this sequence is not "supporting" this subsequence. For example, for the sequence database $D = \{abced, abecf, agbch, abijc, aklc\}$, the support of subsequence abc is 1 if max_gap equals 0 and the support is 3 if max_gap equals 1.

   The gap constraint with $max\_gap = 0$ means that no statement insertiond or deletions are allowed in copy-paste, whereas the gap constraint with $max\_gap = 1$ or $max\_gap = 2$ means that one or two statement insertions/deletions are tolerated in copy-paste.
2. **Matching frequent subsequences to copy-pasted segments.** The original CloSpan algorithm outputs only frequent subsequences and their corresponding support values, but not the supporting sequences that contain them. To identify copy-pasted code, we need to output the supporting sequences for each frequent subsequence.

   We enhanced CloSpan to address this problem. When CP-Miner generates a frequent subsequence, it maintains a list of IDs of its supporting sequences. In the above example, CP-Miner outputs two frequent subsequences: (67641265) and (133872016, 133872016, 82589171), each with their supporting sequence IDs, based on which the locations of the corresponding basic copy-pasted segments (file name and line numbers) can be identified.

3.1.3 Composing Larger Copy-Pasted Segments

Since every sequence fed to the mining algorithm represents a basic block, a basic copy-pasted segment may only be a part of a larger copy-pasted segment. Therefore, it is necessary to combine a basic copy-pasted segment with its neighbors to construct a larger one, if possible.

The composition procedure is very straightforward. CP-Miner maintains a candidate set of copy-paste groups which initially includes all of the basic copy-pasted segments that survive the pruning procedure described in Section 3.1.4. For each copy-paste group, CP-Miner checks their neighboring code segments to see if they also form a copy-paste group. If so, the two groups are combined together to form a larger one. This larger copy-paste group is checked against the pruning procedure. If it can survive the pruning process, it is added to the candidate set and the two smaller ones are removed. Otherwise, the two smaller ones still remain in the set and are marked as "nonexpandable." CP-Miner repeats this process until all groups in the candidate set are nonexpandable.

3.1.4 Pruning False Positives

It is possible that copy-pasted segments discovered by the mining algorithm or the composition process may contain false positives. The main cause of false positives is the tokenization of identifiers (variable, function, type, etc.) in order to tolerate identifier-renaming in copy-paste. Since identifiers of the same type are mapped into the same token, it is possible to identify false copy-pasted segments. For example, all statements similar to $x = y + z$ would have the same hash value, which can introduce many false positives. To prune false positives, CP-Miner has applied several techniques to both basic and composed copy-pasted segments. The pruning techniques are described below.

1. **Pruning unmappable segments.** This technique is used to prune false positives introduced by the tokenization of identifiers. This is based on the observation that if a programmer copy-pastes a code segment and then renames an identifier, he/she would most likely rename this identifier in all its occurrences in the new copy-pasted segment. Therefore, we can build an identifier mapping that maps old names in one segment to their corresponding new ones in the other segment that belongs to the same copy-paste group. In the example shown in Fig. 2, variable *prom_phys_total* is changed into *prom_prom_taken* (except the bug on line 117). The original source identifiers, instead of tokens, are used here for the mapping from one code segment to the other. This source code-level information obtained from the parsing phase is maintained together with each hashed sentence stored in our sequence database.

   A mapping scheme is consistent if there are very few conflicts that map one identifier name to two or more different new names. If no consistent identifier mapping can be established between a pair of copy-pasted segments, they are likely to be false positives.

   To measure the degree of conflict, CP-Miner uses a metric called *ConflictRatio*, which records the conflict ratio for an identifier mapping between two candidate copy-pasted segments. For example, variable A from segment 1 is changed to multiple identifiers (usually just in one or two identifiers) in segment 2. From these multiple identifiers, we first pick out the one that has the largest number of occurrences in A's corresponding positions. Suppose this identifier is a. Then, we calculate what percentage of A in segment 1 is NOT mapped to a in segment 2, where conflict happens. If A is mapped to a in 75 percent of its occurrences in segment 2, but 25 percent of its occurrences is changed into other variables, the *ConflictRatio* of mapping A from segment 1 to segment 2 is 25 percent. Similarly, if only 25 percent of A in segment 1 is mapped to a in segment 2, the *ConflictRatio* of mapping A from segment 1 to segment 2 is 75 percent. Here, the *ConflictRatio* is asymmetric among the code segment pair, so CP-Miner calculates the values in both directions of mapping. The *ConflictRatio* for the whole mapping scheme between these two segments is the weighted sum of *ConflictRatio* of the mapping for each unique identifier. The weight for an identifier A in a given code segment is the fraction of occurrences of A over the total occurrences of all identifiers. If *ConflictRatio* for two candidate copy-pasted segments (in either one of the mapping directions) is higher than a predefined threshold, these two code segments are filtered as false positives. In our experiments, we set the threshold to be 60 percent.

2. **Pruning tiny segments.** Our mining algorithm may find tiny copy-pasted segments that consist of only one to two simple statements. If such a tiny segment cannot be combined with neighboring segments to compose a larger segment, it is removed from the candidate set. This is based on the observation that copy-pasted segments are usually not very small because programmers cannot save much programming effort by copy-pasting tiny code segments.

   CP-Miner measures the size of a segment by the number of tokens in it. This metric is more appropriate than the number of statements because the length of statements is highly variable. If a single statement is very complicated with many tokens, it is still possible for programmers to copy-paste it.

   To prune tiny segments, CP-Miner uses a tunable parameter called *min_size*. If the number of tokens in a pair of copy-pasted segments is fewer than *min_size*, this pair is removed.

3. **Pruning overlapped segments.** The concatenation approach for constructing larger segments will inevitably lead to many segments which overlap. If a pair of candidate copy-pasted segments overlap with each other, they are considered false positives. To avoid such false positives, CP-Miner stops extending the pair of copy-pasted segments once they overlap. For some program structures, such as the *switch* statement, which contain many pairs of self-similar segments, pruning overlapped segments can avoid most of the false positives in *switch* statements.

4. **Pruning segments with large gaps.** Besides the mining procedure for basic copy-pasted segments, the gap constraint is also applied to composed ones. When two neighboring segments are combined, the maximum gap of the newly composed large segment may become larger than a predefined threshold, *max_total_gap*. If this is true, the composition is invalid. So, the newly composed one is not added into the candidate set and the two smaller ones are marked as nonexpandable in the set.

   Of course, even after such rigorous pruning, false positives may still exist. However, we have manually examined 100 random copy-pasted segments reported by CP-Miner for Linux, and only a few false positives (eight) are found. Therefore, the *precision* of our algorithm is around 92 percent. We can only manually examine each identified copy-pasted segment because there are no traces that record the programmers' copy-paste activity during the development of the software.

3.1.5 Computational Complexity of CP-Miner

CP-Miner can extract copy-pasted code directly from a single software with total complexity of $O(n^2)$ in the worst case (where n is the number of lines of code) and the optimizations further improve its efficiency in practice. For example, CP-Miner can identify more than 150,000 copy-pasted segments from 3-4 million lines of code in less than 20 minutes, as shown in our results in Section 5.3. In CP-Miner, we break very large basic blocks into small

TABLE 1
Identifier Mapping in Fig. 1a Example

Identifiers in segment I (line 92–99)	Identifiers in segment II (line 111–118)
iter (9)	iter (9)
num_reg (1)	num_reg (1)
prom_phys_total (4)	prom_prom_taken (3); prom_phys_total (1)
prom_reg_memlist (2)	prom_reg_memlist (2)

The number after each identifier indicates the number of occurrences.

blocks with at most 30 statements before feeding them to the mining algorithm. Therefore, the depth of the search tree is at most 30. With this constraint of the search tree, the mining complexity of CP-Miner is $O(n^2)$ in the worst case, as described in Section 2.3. Furthermore, the optimizations described in Section 2.3 make it more efficient in both time and space than the worst case.

3.2 Detecting Copy-Paste Related Bugs

As we have mentioned in Section 1, one of the main causes of copy-paste related bugs is that programmers forget to modify identifiers consistently after copy-pasting. Once we get the mapping relationship between identifiers in a pair of copy-pasted segments (see Section 3.1.4), we can find the inconsistency and report these copy-paste related bugs. Table 1 shows the identifier mapping for the example described in Section 1.

For an identifier that appears more than once in a copy-pasted segment, it is consistent when it always maps to the same identifier in the other segment. Similarly, it is inconsistent when it maps to multiple identifiers. In Table 1, we can see that $prom\_phys\_total$ is mapped inconsistently because it maps to $prom\_prom\_taken$ three times and $prom\_phys\_total$ once. All the other variable mappings are consistent.

Unfortunately, inconsistency does not necessarily indicate a bug. If the amount of inconsistency is high, it may indicate that the code segments are not copy-pasted. Section 3.1.4 describes how we prune unmappable copy-pasted segments.

Therefore, the challenge is to decide when an inconsistency is likely to be a bug rather than a false positive of copy-paste. To address this challenge, we need to consider the programmers' intentions. In practice, following different intentions may cause different identifier inconsistency as follows (take identifier A in segment 1 copied to segment 2 as example): 1) The programmer intends to change every A to another identifier, say a. He/she conducts this change in most places, but forgets some corner positions. This type of mistake happens quite often. 2) The programmer intends to maintain exactly the same A in segment 2, but he carelessly changes some A to other identifiers. 3) The programmer intends to change A to several different identifiers in segment 2 and inconsistency happens during such modification.

Analyzing the three cases above, we found that if a programmer changes an identifier in most places but forgets to change it in a few places, the unchanged identifier is very likely to be a bug. In other words, "forget-to-change" is more likely to be a bug than an intentional "change." For example, if, in *some* cases, an identifier A is mapped into a and, in other cases, it is mapped into a' (both a and a' are different from A), it is a bug with a low probability because the programmer is likely to *intentionally* change A to several names due to functionality requirements. On the other hand, if A is changed into a in *most* cases but remains unchanged only in a *few* cases, the unchanged places are likely to be bugs.

In order to verify this intuition, we also tried to implement CP-Miner to report those copy-pasted segments where programmers changed an identifier in some occurrences but most of the occurrences remained unchanged. We rank the reports based on the "changed ratio," similar to the $UnchangedRatio$ described in the following. By manually verifying the top 100 reports for Linux, we only found two of them are real bugs. The false positive rate is much higher than the first case (see Section 5.2). To this end, the current version of CP-Miner only addresses the first case because it is more likely to happen than the second case and relatively easier to detect than the third case.

Based on the above observation, CP-Miner reexamines each nonexpandable copy-paste group after running through the pruning and composing procedures. For each pair of copy-pasted segments, it uses a metric called $UnchangedRatio$ to detect bugs in an identifier mapping. We define

$$UnchangedRatio = \frac{NumUnchanged}{NumTotal},$$

where $NumUnchanged$ is the number of occurrences in which a given identifier is unchanged and $NumTotal$ is the total number of occurrences of this identifier in a given copy-pasted segment. $UnchangedRatio$ can be any value between 0 and 1 (inclusive). Especially, $UnchangedRatio = 0$ means that all occurrences of the identifier have been changed, while $UnchangedRatio = 1$ means that all of its occurrences remain unchanged. Therefore, *the lower the* $UnchangedRatio$*, the more likely it is a bug* unless $UnchangedRatio = 0$. Note that $UnchangedRatio$ is different from $ConflictRatio$. The former only measures the ratio of unchanged occurrences, whereas the latter measures the ratio of conflicts. In the example shown in Table 1, $UnchangedRatio$ for $prom\_phys\_total$ is 0.25, whereas all other identifiers have $UnchangedRatio = 1$.

CP-Miner uses a threshold for $UnchangedRatio$ to detect bugs, denoted as $UnchangedRatioThreshold$. If $UnchangedRatio$ for an identifier is not zero and not larger than $UnchangedRatioThreshold$, the unchanged occurrences are reported as bugs. Notice that $UnchangedRatio$ for a pair of segments has two different directions. Because we cannot tell which segment is the original one, CP-Miner calculates in both directions. When CP-Miner reports a bug, the corresponding identifier mapping information is also provided to programmers to help in debugging. In the example shown in Table 1, identifier $prom\_phys\_total$ on line 117 is reported as a bug.

It is possible to further extend CP-Miner's bug detection engine. For example, it might be useful to exploit variable

TABLE 2
Software Evaluated in Our Experiments

Software	version	#files	#LOC
Linux	2.6.6	6,497	4,365,124
FreeBSD	5.2.1	7,114	3,299,622
Apache	2.0.49	479	223,886
PostgreSQL	7.4.2	553	458,058

correlations. Assume variable A always appears in close range to another variable B and a always appears very close to b. So, if, in a pair of copy-pasted segments, A is renamed to a, B then should be renamed to b with high confidence. Any violation of this rule may indicate a bug. But, the current version of CP-Miner has not exploited this idea, which remains as future work.

4 Methodology

We have evaluated the effectiveness of CP-Miner with large software, including Linux, FreeBSD, Apache Web server, and PostgreSQL, in our experiment. The number of files (only C files) and the number of lines of code (LOC) for the software are shown in Table 2.

We set the thresholds used in CP-Miner as follows: The minimum copy-pasted segment size min_size is 30 tokens. We also vary the gap constraints: 1) When $max\_gap = 0$, CP-Miner only identifies copy-pasted code with identifier-renaming; 2) when $max\_gap = 1$ and $max\_total\_gap = 2$, it means that CP-Miner allows copy-pasted segments with insertion and deletion of one statement between any two consecutive statements, and a total of two statement insertions and deletions in the whole segment. We use setting 2) by default because it permits a reasonable size of modification after copy-paste and will not cause too many false positives.

We define $CP\%$ to measure the percentage of copy-paste in given software (or a given module):

$$CP\% = \frac{\#\text{LOC in copy-pasted segments}}{\#\text{LOC in the software suite or the module}} \times 100\%.$$

In our experiments in identifying copy-pasted code in the four software suites above, we also compare CP-Miner with a recently proposed tool, called *CCFinder* [5]. Similarly to our tool, CCFinder also tokenizes identifiers, keywords, constant, operators, etc. But, differently from our tool, it uses a suffix tree algorithm instead of a data mining algorithm. Therefore, it cannot tolerate statement insertions and deletions in copy-pasted code. Our results show that CP-Miner detects 17-52 percent more copy-pasted code than CCFinder. In addition, CCFinder does not filter incomplete, tiny copy-pasted segments, which are very likely to be false positives. CCFinder does not detect copy-paste related bugs, so we cannot compare this functionality between them.

In our experiments, we ran CP-Miner and CCFinder on an Intel Xeon 2.4GHz machine with 2GB memory.

TABLE 3
The Number of Copy-Pasted Segments and $CP\%$

Software	$max\_gap = 0$		$max\_gap = 1$	
	#Segments	$CP\%$	#Segments	$CP\%$
Linux	122,282	15.3%	198,605	22.3%
FreeBSD	101,699	14.9%	153,230	20.4%
Apache	4,155	13.1%	6,196	17.7%
PostgreSQL	12,105	16.5%	16,662	22.2%

5 CP-Miner Basic Results

We first present the basic results of CP-Miner in this section, including the number of copy-pasted segments, the number of detected copy-pasted bugs, CP-Miner overhead, comparison with CCFinder, and effects of threshold setting. The statistical results of copy-paste characteristics in Linux and FreeBSD will be presented in Section 6.

5.1 Overall Results

5.1.1 Detecting Copy-Pasted Code

CP-Miner has found a significant number of copy-pasted segments in the evaluated software. With $max\_gap = 2$, the total amount of copy-paste accounts for 17.7-22.3 percent of the source code in these software suites, with Apache being the lowest, 17.7 percent, and Linux being the highest, 22.3 percent. Table 3 shows the numbers of copy-pasted segments and $CP\%$ for these software suites. As shown in this table, in Linux and FreeBSD, there are more than 100,000 and 120,000 copy-pasted segments without any statement insertion ($max\_gap = 0$), which account for about 15 percent of the source code. We have manually examined the identified copy-pasted segments in one of the Linux modules (file system) and found very few (only three) false positives out of 90 identified copy-pasted segments (with $max\_gap = 1$). The large number of copy-pasted segments motivates a support in integrated development environments (IDEs) such as Eclipse, Microsoft Visual Studio, and NetBeans to maintain copy-pasted code.

Our results also show that a large percentage (30-50 percent) of copy-pasted segments have statement insertions and modifications. For example, when max_gap is 1, CP-Miner finds 62.4 percent more copy-pasted segments in Linux. In FreeBSD, the $CP\%$ increases from 14.9 percent to 20.4 percent when max_gap is relaxed from 0 to 1. These results show that previous tools, including CCFinder, which cannot tolerate statement insertions and modifications, would miss a lot of copy-paste.

By increasing max_gap from 1 to 2 or higher, we can further relax the gap constraint. Due to space limitations, we do not show those results here. Also, the number of false positives will increase with max_gap. Our manual examination results with the Linux file system module indicate that false positives are low with $max\_gap = 1$, and relatively low with $max\_gap = 2$.

5.1.2 Detecting Copy-Paste Related Bugs

CP-Miner has also reported many copy-paste related bugs in the evaluated software. Since the bugs reported by CP-Miner may not be real bugs, we verify each one manually and then

TABLE 4
Copy-Paste Related Bugs Reported by CP-Miner ($UnchangedRatioThreshold = 0.4$) and Bugs Verified by Us with High Confidence, Most of Which Were Confirmed and Fixed by Corresponding Developers after We Reported

Software	bugs reported	bugs verified	false positives			unconfirmed
			(1)	(2)	(3)	
Linux	421	49	151	41	57	123
FreeBSD	443	31	307	41	30	34
Apache	17	5	3	1	6	2
PostgreSQL	74	2	13	10	43	6

The false alarms include three categories: 1) incorrectly matched segments, 2) exchangeable orders, and 3) others. The first two categories can be pruned, which remains as our future work.

report to the corresponding developer community those bugs that we suspect to be real bugs with high confidence. The numbers of bugs detected by CP-Miner and verified bugs are shown in Table 4. The results are achieved by setting the $UnchangedRatioThreshold$ to be 0.4.

Both Linux and FreeBSD have many copy-paste related bugs. So far, we have verified 49 and 32 bugs in the latest versions of Linux and FreeBSD. Most of these bugs had never been reported before. We have reported these bugs to the kernel developer communities. *Most of the Linux bugs have been confirmed and fixed in the following releases by Linux kernel developers and the others are still in the process of being confirmed.*

Since Apache and PostgreSQL are much smaller compared to Linux and FreeBSD, CP-Miner found many fewer copy-paste related bugs. We have verified six bugs for Apache and two bugs for PostgreSQL with high confidence. *One bug in Apache was immediately fixed by the Apache developers after we reported it to them.*

5.2 False Positives in Bug Detection

Table 4 also shows the number of false positives reported by CP-Miner in bug detection. These false positives are mostly caused by the following two major reasons and can be further pruned in our future work:

1. **Incorrectly matched copy-pasted segments.** In some copy-pasted segments that contain multiple "*case/if*" blocks, there are many possible combinations for these contiguous copy-pasted blocks to compose larger ones. Since CP-Miner simply follows the program order to compose larger copy-pastes, it is likely that a wrong combination might be chosen. As a result, identifiers are compared between two incorrectly matched copy-pasted segments, which results in false positives.

   These false positives can be pruned if we use more semantic information of the identifiers in these segments. The segments with a number of "*case/if*" blocks usually contain a lot of constant identifiers, but our current CP-Miner treats them as normal variable names. If we use the information of these constants to match "*case/if*" blocks when composing larger copy-pasted segments, it can reduce the number of incorrectly matched segments and most false positives can be pruned.

TABLE 5
Execution Time and Memory Space of CP-Miner

Software	$max\_gap = 0$		$max\_gap = 1$	
	Time(s)	Space(MB)	Time(s)	Space(MB)
Linux	770	438	1164	527
FreeBSD	615	334	1155	459
Apache	14	27	15	30
PostgreSQL	32	44	38	57

2. **Exchangeable orders.** In a copy-paste pair, the orders of some statements or expressions can be switched. For example, a segment with several similar statements such as "*a1=b1; a2=b2;*" is the same as "*a2=b2; a1=b1;*." The current version of CP-Miner simply compares the identifiers in a pair of copy-pasted segments in strict order and, therefore, a false alarm might be reported. In Linux, 41 false positives are caused by such exchangeable orders.

   These false positives can be pruned if we relax the strict order comparison by further checking whether the corresponding "changed" identifiers are in the neighboring statements/expressions.

5.3 Time and Space Overheads

CP-Miner can identify copy-pasted code in large-scale software code very efficiently. The execution time of CP-Miner is shown in Table 5. CP-Miner takes 11-20 minutes to identify 101,699-198,605 copy-pasted segments in Linux and FreeBSD, each with 3-4 millions of lines of code. It takes less than 1 minute to detect copy-pasted segments in Apache and PostgreSQL with more than 200,000 lines of code.

CP-Miner is also space-efficient. For example, it takes less than 530MB to find copy-pasted code in Linux. For Apache and PostgreSQL, CP-Miner only consumes 27-57 MB of memory.

5.4 Comparison with CCFinder

We have compared CP-Miner with CCFinder [5]. CCFinder has similar execution time as CP-Miner, but CP-Miner discovers many more copy-pasted segments. In addition, CCFinder cannot detect copy-paste related bugs. As we explained in Section 4, CCFinder allows identifier-renaming, but not statement insertions. In addition, CCFinder does not have as rigorous pruning operations as CP-Miner. For example, CCFinder reports many tiny copy-pasted segments fewer than 30 tokens, which are too simple to be worth copy-pasting. In addition, it also includes incomplete statements in copy-pasted segments, which is very unlikely to be the case in practice.

CP-Miner can identify 17-52 percent more copy-pasted code than CCFinder because CP-Miner can tolerate statement insertions and modifications. Table 6 compares the $CP\%$ identified by CP-Miner and CCFinder. The results with CP-Miner are achieved using the default threshold setting ($min\_size = 30$ and $max\_gap = 1$). For fair comparison, we also filter those tiny, incomplete segments from CCFinder's output. The results show that around 25 percent of copy-paste is pruned after filtering.

TABLE 6
$CP\%$ Comparison between CP-Miner and CCFinder

Software	CCFinder	CP-Miner
Linux	14.7%(19.8%)	22.3%
FreeBSD	14.5%(19.6%)	20.4%
Apache	11.8%(15.3%)	17.0%
PostgreSQL	18.5%(23.8%)	21.7%

For CCFinder, the first number is the result after pruning those tiny, incomplete segments, and the second number in $()$ is the result before pruning.

5.5 Effects of Threshold Settings

Segment Size Threshold. Fig. 3 shows the effect of segment-size threshold min_size on $CP\%$. As expected, the $CP\%$ decreases when min_size increases because more copy-pasted segments are pruned. The results also show that the decrement slows down when min_size is in the range of 30-100 (tokens), which indicates that not too many copy-segments' sizes fall in this range. This implies that segments with fewer than 30 tokens are very likely to be false positives, whereas those with more than 40 tokens are very likely to be copy-paste.

Unchanged Ratio Threshold. Fig. 4 shows the effect of threshold $UnchangedRatioThreshold$ on the number of bugs reported. Since $UnchangedRatio \geq 0.5$ means that at least half of the identifiers are not changed after copy-paste, these unchanged identifiers are unlikely "forget-to-change" and, so, it cannot indicate a copy-paste related error. Therefore, we only show the errors with $UnchangedRatioThreshold < 0.5$.

As expected, more errors are reported by CP-Miner when $UnchangedRatioThreshold$ increases. Specifically, the number of errors reported increases gradually when the threshold is less than 0.25 and then increases sharply when the threshold $\in (0.25, 0.35)$. We found that most of the errors with high $UnchangedRatio$ turn out to be false bugs during our verification. For example, CP-Miner reports many errors where only one out of three identifiers is unchanged ($UnchangedRatio = 0.33$). However, it cannot strongly support that it is a copy-paste related bug. In order to prune such false bugs, we can further analyze the identifiers in the context of the copy-pasted segments (e.g., the whole function). We leave this improvement as future work.

6 Statistics of Copy-Paste in OS Code

This section presents the statistical results on copy-paste characteristics in large-scale software code. Our results include the distribution of copy-pasted segments across different group sizes, segment sizes, granularity, amount of changes, modules, and versions.

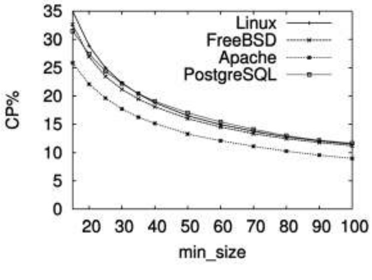


Fig. 3. Effects of min_size on $CP\%$.

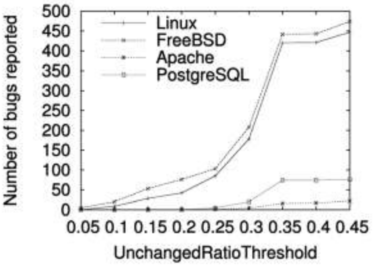


Fig. 4. Effects of $UnchangedRatioThreshold$ on errors reported. Since Linux and FreeBSD are much larger than Apache and PostgreSQL, the numbers of bugs in the former two applications are also significantly more than the latter two, especially when the threshold increases.

6.1 Copy-Pasted Segment Size and Granularity

Fig. 5 illustrates the distribution of copy-pasted segments with different sizes (in terms of the number of statements). The results show that most (60-64 percent) copy-pasted segments are not very large, with only 5-16 statements. Only a few (0.2-5.0 percent) copy-pasted segments have more than 64 statements. In particular, Fig. 5a shows that most (35-40 percent) copy-paste groups contain five to eight statements in each segment. Fig. 5b shows similar characteristics: Copy-pasted segments with five to eight statements cover about 7-10 percent of the source code.

Fig. 6 shows the distribution of copy-paste group size. About 60 percent of copy-paste groups contain only two segments, which indicates that there are only two copies (original and replicated) for most copy-pasted code. But still, 40 percent of the copy-pasted code groups contain at least three segments, which indicates that a lot of code is replicated more than once.

We found that 4.0-6.7 percent of copy-pasted segments are copy-pasted more than seven times. If a bug is detected in one of the copies, it is very difficult for programmers to remember fixing the bug in the other seven or more copies. This motivates a tool that can automatically fix other copy-pasted segments once a programmer fixes one segment.

Table 7 shows the number of copy-pasted segments at basic-block and function granularity. Our results show that only 3-11.8 percent of all copy-pasted segments are basic blocks, which indicates that programmers are less likely to copy-paste basic blocks than whole functions because some of them are too simple to be worth copy-pasting.

More interestingly, there are many (11.3-19.2 percent) copy-pasted segments at function granularity. The reason is that many functions provide similar functionality, such as reading data from different types of I/O devices. Those functions can be copy-pasted with some modifications, such as replacing parameters' data types. This motivates some refactoring tools [39] to better maintain these copy-pasted functions.

6.2 Modifications in Copy-Pasted Segments

Fig. 7 shows how many identifiers are changed in copy-pasted segments. Since, in some cases, there are more than two segments in each copy-paste group, we only present

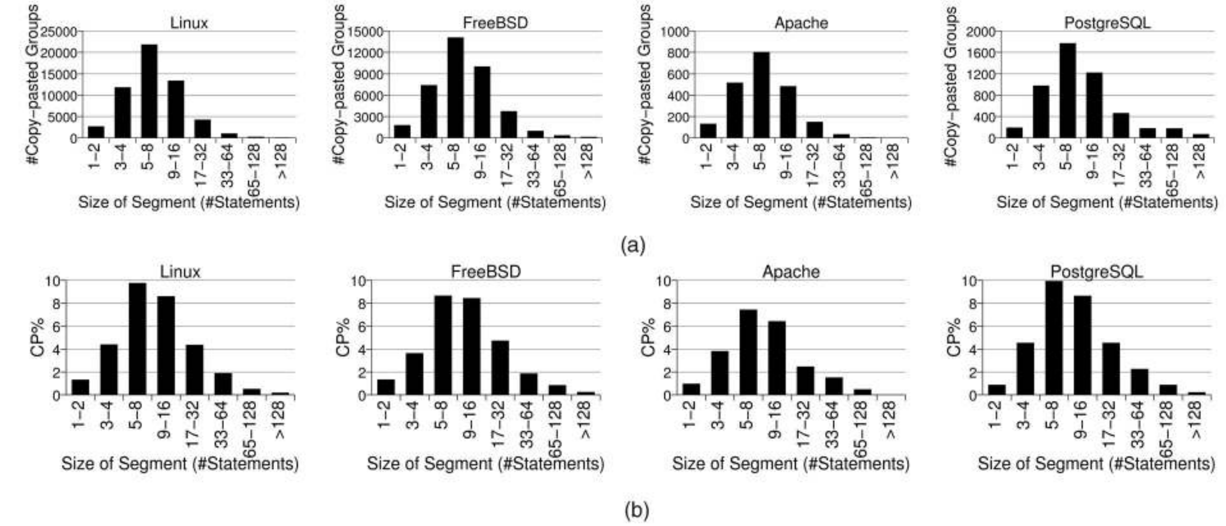


Fig. 5. Size distribution of copy-pasted segments. Due to the overlap of copy-pasted segments that have different segment sizes and also belong to different copy-paste groups, the sum of all $CP\%$ does not equal the overall $CP\%$. (a) The number of copy-paste groups with various segment sizes (number of statements). (b) The $CP\%$ with various segment sizes (number of statements).

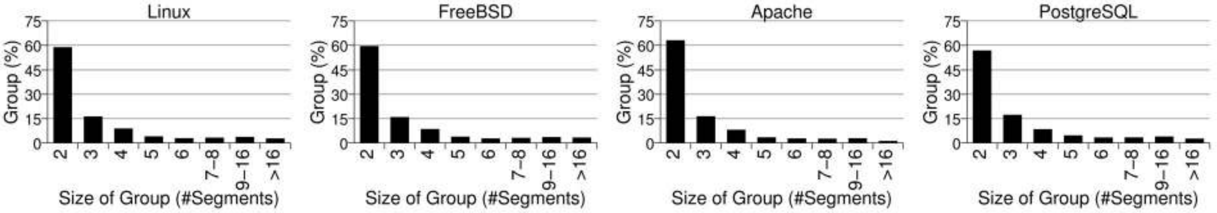


Fig. 6. Copy-paste group size distribution (in terms of the number of segments in each copy-paste group). Each bar represents the percentage of copy-paste groups that contains the corresponding number of segments.

the distribution in the best case: comparing the most similar pair of segments from each copy-paste group. Each bar includes two parts: one with no statement insertion and the other with one statement insertion.

The results indicate that 59-76 percent of copy-pasted segments require identifier-renaming. For example, in Linux, 27 percent of copy-pasted segments are identical and 8 percent of segments are almost identical with only one statement inserted. The remaining 65 percent of the copy-pasted segments in Linux rename at least one identifier. Such results motivate a tool to support consistently renaming identifiers in copy-pasted code.

6.3 Copy-Pasted Code across Modules

Different modules in an application have different characteristics of copy-paste. In this subsection, we analyze the copy-pasted code across different modules in operating system code. We split Linux into nine categories: arch (platform specific), fs (file system), kernel (main kernel), mm (memory management), net (networking), sound (sound device drivers), drivers (device drivers other than networking and sound device), crypto (cryptography), and others (all other code). For FreeBSD, modules are also split into nine categories: sys (kernel sources), lib (system libraries), crypto (cryptography), usr.sbin (system administration commands), usr.bin (user commands), sbin (system commands), bin (system/user commands), gnu, and others.

TABLE 7
Distribution of Copy-Paste Granularity: Numbers and Percentages of Copy-Pasted Segments at Different Granularity

Software	basic block	function
Linux	17,818 (9.0%)	26,744 (13.5%)
FreeBSD	13,999 (9.1%)	17,254 (11.3%)
Apache	733 (11.8%)	1,189 (19.2%)
PostgreSQL	494 (3.0%)	2,280 (13.7%)

Note here the percentage is not $CP\%$. It is calculated by comparing to the total number of copy-pasted segments.

6.3.1 Distribution of Copy-Pasted Code in Modules

Fig. 8 shows the number and $CP\%$ of copy-pasted segments in different modules of Linux and FreeBSD. The $CP\%$ is computed based on the size of each corresponding module, instead of the entire software suite.

Fig. 8a shows that most copy-pasted code in Linux and FreeBSD is located in one or two main modules. For example, modules "drivers" and "arch" account for 71 percent of all copy-pasted code in Linux and module "sys" accounts for 60 percent in FreeBSD. This is because many drivers are similar and it is much easier to modify a copy-paste of another driver than writing one from scratch. For example, the SCSI drivers in Linux for a series of QLogic devices (ISP2100, ISP2200, ISP2300, ISP2322, ISP6312, ISP6322, etc.) are very similar to each other, each

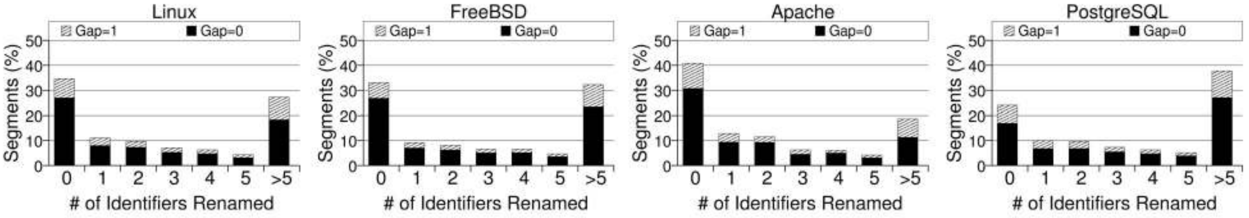


Fig. 7. Distribution of identifiers changed in copy-pasted segments. Each bar represents the percentage of segments that have the corresponding number of renamed identifiers. Each bar has two parts: "$Gap = 0$" and "$Gap = 1$" represent the copy-pasted segments with no and one statement modifications, respectively.

of which is based on the driver for the previous series of device with a little changes.

Fig. 8b shows that a large percentage (20-28 percent) of the code in Linux is copy-pasted in the modules "arch," "crypto," and the device driver modules including "net," "sound," and "drivers." The "arch" module has a lot of copy-pasted code because it has many similar submodules for different platforms. The device driver modules contain a significant portion of copy-pasted code because many devices share similar functionalities. Additionally, "crypto" is a very small module (less than 10,000 LOC), but the main cryptography algorithms consist of a number of similar computing steps, so it contains a lot of copy-pasted code. Our results indicate that more attention should be paid to these modules because they are more likely to contain copy-paste related bugs.

In contrast, the modules "mm" and "kernel" contain much less copy-pasted code than others, which indicates that it is rare to reuse code in kernels and memory management modules.

6.3.2 Copy-Pasted Code within/across Modules

The code in a module can be copy-pasted within the module itself or from other modules. Table 8 shows how much code is copy-pasted within the module itself or across different modules.

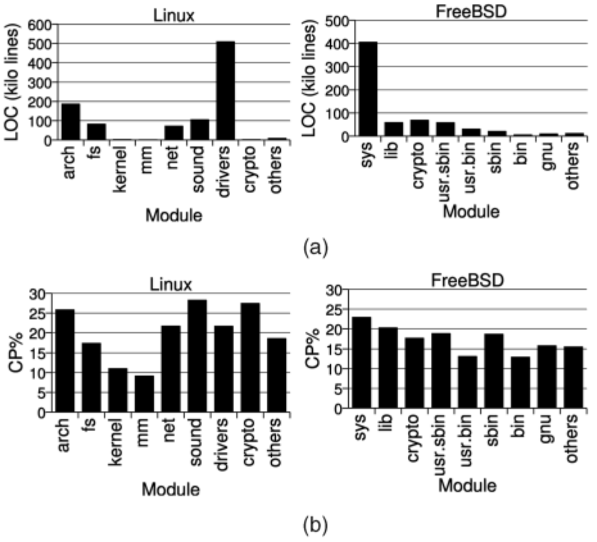


Fig. 8. Copy-pasted code in different modules. (a) The number of copy-pasted lines. (b) $CP\%$.

Most copy-pasted code is within the same module, as indicated by the bold numbers in Table 8. The percentage of copy-pasted code within a module is usually greater than 15 percent and it is greater than 20 percent in some cases. Meanwhile, the percentage of code that is copy-pasted across different modules is usually lower than 4 percent, much lower than the percentage within a module.

The exceptional case is that 4.6 percent of the code in the "sound" module in Linux is copy-pasted from "drivers." The reason is that the "sound" module was originally one of the submodules in the "drivers" module before it was separated since version 2.5.5. Therefore, the "sound" module still contains a lot of duplicate code from the "drivers" module.

6.4 Copy-Paste Evolution

Fig. 9 shows that the amount of copy-pasted code increases as the operating system code evolves. For example, Fig. 9a shows the amount of copy-pasted code in Linux from version 1.0 to 2.6.6 over time. As Linux's code size increases from 141,000 to 4.4 million lines, copy-pasted code also keeps increasing from 23,000 to 975,000 lines.

In terms of $CP\%$, the percentage of copy-pasted code also steadily increases along software evolution. For example, Fig. 9a shows that $CP\%$ in Linux increases from 16.2 percent to 22.3 percent in Linux from version 1.0 to 2.6.6 and Fig. 9c shows that $CP\%$ in FreeBSD increases from 17.5 percent to 21.7 percent from version 2.0 to 4.10. However, the $CP\%$ remains relatively stable over the several recent versions for both Linux and FreeBSD. For example, the $CP\%$ for FreeBSD has been staying at around 21-22 percent since version 4.0.

Most of the growth of $CP\%$ comes from a few modules, including "drivers" and "arch" in Linux and "sys" in FreeBSD. Fig. 9b shows copy-pasted code in the module "drivers" individually through multiple versions. The percentage of copy-pasted code increases more rapidly in this module than in the entire software suite. For example, in version 1.0, the $CP\%$ is only 11.9 percent in this module, but it increases to 20.4 percent in version 2.2.0. This is probably because Linux supports more and more similar device drivers during this period.

7 CONCLUSIONS

This paper presents a tool called CP-Miner that uses data mining techniques to efficiently identify copy-pasted code in large software including operating systems and also

TABLE 8
Copy-Paste Code within a Module and across Modules: (a) Linux 2.6.6 and (b) FreeBSD 5.2.1

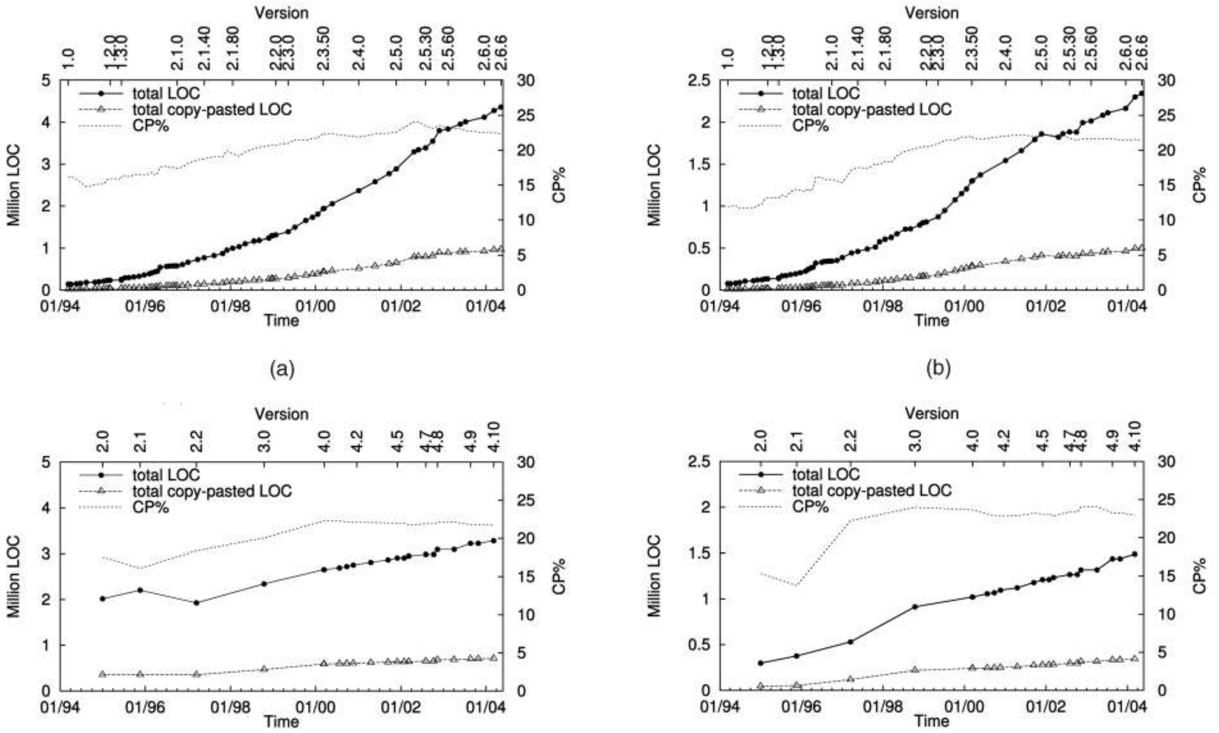
Module (LOC)	arch	fs	kernel	mm	net	sound	drivers	crypto	others
arch (724858)	**25.1**	1.4	0.5	0.3	1.1	1.3	3.2	0.1	0.8
fs (475946)	1.4	**16.5**	0.6	0.5	1.7	1.2	2.2	0.0	0.7
kernel (30629)	3.0	1.8	**7.9**	0.6	2.3	1.6	2.8	0.1	0.8
mm (23490)	2.6	2.2	0.8	**6.2**	1.7	1.1	2.0	0.0	0.7
net (334325)	1.8	2.5	1.1	0.7	**20.7**	2.1	3.7	0.1	1.0
sound (373109)	2.3	2.0	1.0	0.6	2.2	**27.4**	4.6	0.2	1.1
drivers (2344594)	2.3	1.7	0.6	0.4	1.8	2.0	**21.4**	0.1	0.6
crypto (9157)	2.3	2.2	0.3	0.1	1.1	1.5	2.5	**26.1**	2.2
others (49016)	3.8	1.9	0.8	0.4	1.7	1.5	2.6	0.3	**15.2**

(a)

Module (LOC)	sys	lib	crypto	usr.sbin	usr.bin	sbin	bin	gnu	others
sys (1767368)	**22.5**	1.5	1.1	1.5	1.2	1.0	0.3	0.6	0.8
lib (291132)	3.3	**18.1**	1.8	1.4	1.2	0.7	0.3	0.5	0.9
crypto (392020)	2.1	1.6	**16.7**	1.5	1.2	1.0	0.4	0.9	0.9
usr.sbin (310949)	3.5	2.5	2.2	**17.7**	3.8	2.8	1.0	1.6	2.5
usr.bin (236952)	2.6	1.8	1.7	3.4	**11.9**	2.2	1.2	1.1	1.9
sbin (112284)	3.1	2.0	1.7	3.5	3.1	**16.9**	1.2	1.1	2.2
bin (47008)	1.3	1.0	2.0	2.0	2.3	1.8	**10.9**	0.8	1.6
gnu (64996)	2.8	1.7	2.0	1.7	1.6	0.9	0.7	**14.5**	1.1
others (76913)	2.9	2.4	2.0	3.3	3.1	2.6	1.0	1.2	**12.7**

(b)

Each number in the table represents the $CP\%$ of code copy-pasted from another module. For example, in (a), the number at row "arch" and column "arch" represents that 25.1 percent of the code in module "arch" is copy-pasted within the module itself; the number at row "arch" and column "drivers" represents that 3.2 percent of the code in module "arch" is copy-pasted from/to another module "drivers." Note that these tables are asymmetric because $CP\%$ is related to the size of the row element module.



(a)

(b)

(c)

(d)

Fig. 9. Copy-pasted code in Linux and FreeBSD through various versions. The x-axis (version number) is drawn in time scale with the corresponding release time. The versions of Linux we analyze are from 1.0 to the current version 2.6.6. The versions of FreeBSD include the main branch from 2.0 to 4.10. (a) Linux 1.0-2.6.6. (b) "Drivers" in Linux. (c) FreeBSD 2.0-4.10. (c) "Sys" in FreeBSD.

detects copy-paste related bugs. Specifically, it takes less than 20 minutes for CP-Miner to identify 190,000 and 150,000 copy-pasted segments that account for 20-22 percent of the source code in Linux and FreeBSD. Moreover, CP-Miner has detected 49 and 31 copy-paste related bugs with an acceptable false positive rate in the latest versions of Linux and FreeBSD, respectively. Compared to CCFinder [5], CP-Miner finds 17-52 percent more copy-pasted segments because it can tolerate statement insertions and modifications in copy-paste. In addition, we have shown some interesting characteristics of copy-pasted codes in Linux and FreeBSD, including distribution of copy-paste by different segment sizes (number of lines of code), granularity (basic blocks and functions), and modification. We also analyze copy-paste across different modules and various software versions.

Our results indicate that maintaining copy-pasted code would be very useful for programmers because it is commonly used in large-scale software such as operating system code and it can easily introduce hard-to-detect bugs. We hope our study motivates IDEs such as Eclipse, Microsoft Visual Studio, and NetBeans to provide functionality to maintain copy-pasted code and automatically detect copy-paste related bugs.

Even though CP-Miner focuses only on "forget-to-change" bugs caused by copy-paste, copy-paste can introduce many other types of bugs. For example, after the copy-paste operation, the programmer forgets to add some statements that are specific to the new copy-pasted segment. However, such bugs are hard to detect because detecting them relies on semantic information. It is impossible to guess what the programmer would want to insert or modify. Another type of copy-paste related bug is caused by programmers forgetting to fix a known bug in all copy-pasted segments. They may fix only one or two segments, but forget to change it in the others. Our tool CP-Miner can detect simple cases of this type of error. But, if the fix is too complicated, CP-Miner would miss the bug because the modified code segment becomes too different from the other copies to be identified as copy-paste. To overcome this problem would require support from IDEs such as Eclipse, Microsoft Visual Studio, NetBeans, etc.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers and James Larus (Microsoft Research Lab) for their invaluable feedback. They appreciate Professor Jiawei Han and his group for their CloSpan mining algorithm. This work was supported by an IBM Faculty Award, US National Science Foundation (NSF) CNS-0347854 (career award), NSF CCR-0305854 grant, and NSF CCR-0325603 grant. The experiments were conducted on equipment provided through the IBM SUR grant. An earlier version of this paper [1] appeared in the *Proceedings of the Sixth Symposium on Operating System Design and Implementation*.

REFERENCES

[1] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: A Tool for Finding Copy-Paste and Related Bugs in Operating System Code," *Proc. Symp. Operating System Design and Implementation,* pp. 289-302, 2004.

[2] B.S. Baker, "On Finding Duplication and Near-Duplication in Large Software Systems," *Proc. Second Working Conf. Reverse Eng.,* p. 86, 1995.

[3] S. Ducasse, M. Rieger, and S. Demeyer, "A Language Independent Approach for Detecting Duplicated Code," *Proc. Int'l Conf. Software Maintenance,* pp. 109-118, 1999.

[4] C. Kapser and M.W. Godfrey, "Toward a Taxonomy of Clones in Source Code: A Case Study," *Evolution of Large-Scale Industrial Software Applications,* Sept. 2003.

[5] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A Multi-linguistic Token-Based Code Clone Detection System for Large Scale Source Code," *IEEE Trans. Software Eng.,* vol. 28, no. 7, pp. 654-670, July 2002.

[6] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees," *Proc. Int'l Conf. Software Maintenance,* p. 368, 1998.

[7] A. Chou, J. Yang, B. Chelf, S. Hallem, and D.R. Engler, "An Empirical Study of Operating System Errors," *Proc. Symp. Operating Systems Principles,* pp. 73-88, 2001.

[8] "Linux Kernel Mailing List," http://lkml.org, year?

[9] A. Chou, B. Chelf, D.R. Engler, and M. Heinrich, "Using Meta-Level Compilation to Check FLASH Protocol Code," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating System,* pp. 59-70, 2000.

[10] D. Engler and K. Ashcraft, "RacerX: Effective, Static Detection of Race Conditions and Deadlocks," *Proc. ACM Symp. Operating Systems Principles,* pp. 237-252, 2003.

[11] S. Hallem, B. Chelf, Y. Xie, and D. Engler, "A System and Language for Building System-Specific, Static Analyses," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 69-82, 2002.

[12] M. Musuvathi, D. Park, A. Chou, D.R. Engler, and D.L. Dill, "CMC: A Pragmatic Approach to Model Checking Real Code," *Proc. Symp. Operating Systems Design and Implementation,* Dec. 2002.

[13] R. Hastings and B. Joyce, "Purify: Fast Detection of Memory Leaks and Access Errors," *Proc. Winter USENIX Conf.,* pp. 158-185, Dec. 1992.

[14] N. Nethercote and J. Seward, "Valgrind: A Program Supervision Framework," *Proc. Third Workshop Runtime Verification,* 2003.

[15] J. Condit, M. Harren, S. McPeak, G.C. Necula, and W. Weimer, "CCured in the Real World," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 232-244, 2003.

[16] S. Grier, "A Tool that Detects Plagiarism in Pascal Programs," *Proc. 12th SIGCSE Technical Symp. Computer Science Education,* pp. 15-20, 1981.

[17] H.T. Jankowitz, "Detecting Plagiarism in Student Pascal Programs," *Computer J.,* vol. 31, no. 1, pp. 1-8, 1988.

[18] L. Prechelt, G. Malpohl, and M. Philippsen, "Finding Plagiarisms among a Set of Programs with JPlag," *J. Universal Computer Science,* vol. 8, no. 11, pp. 1016-1038, Nov. 2002.

[19] A. Aiken, "Moss: A System for Detecting Software Plagiarism," http://www.cs.berkeley.edu/aiken/moss.html, year?

[20] S. Schleimer, D.S. Wilkerson, and A. Aiken, "Winnowing: Local Algorithms for Document Fingerprinting," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 76-85, 2003.

[21] B.S. Baker, "A Program for Identifying Duplicated Code," *Computing Science and Statistics,* vol. 24, pp. 49-57, 1992.

[22] J.H. Johnson, "Substring Matching for Clone Detection and Change Tracking," *Proc. Int'l Conf. Software Maintenance,* pp. 120-126, 1994.

[23] J.H. Johnson, "Identifying Redundancy in Source Code Using Fingerprints," *Proc. Conf. Centre for Advanced Studies on Collaborative Research,* Oct. 1993.

[24] R. Komondoor and S. Horwitz, "Using Slicing to Identify Duplication in Source Code," *Proc. Eighth Int'l Symp. Static Analysis,* 2001.

[25] K. Kontogiannis, M. Galler, and R. DeMori, "Detecting Code Similarity Using Patterns," *Working Notes Third Workshop AI and Software Eng.: Breaking the Toy Mold,* 1995.

[26] J. Krinke, "Identifying Similar Code with Program Dependence Graphs," *Proc. Eighth Working Conf. Reverse Eng.,* 2001.

[27] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics," *Proc. Int'l Conf. Software Maintenance,* p. 244, 1996.

[28] U. Manber, "Finding Similar Files in a Large File System," *Proc. USENIX Winter 1994 Technical Conf.*, pp. 1-10, 1994.

[29] K.W. Church and J.I. Helfman, "Dotplot: A Program for Exploring Self-Similarity in Millions of Lines of Text and Code," *J. Computational and Graphical Statistics*, 1993.

[30] S. Hangal and M.S. Lam, "Tracking Down Software Bugs Using Automatic Anomaly Detection," *Proc. Int'l Conf. Software Eng.*, May 2002.

[31] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A Dynamic Data Race Detector for Multithreaded Programs," *ACM Trans. Computer Systems*, vol. 15, no. 4, pp. 391-411, 1997.

[32] D. Engler, D.Y. Chen, and A. Chou, "Bugs as Inconsistent Behavior: A General Approach to Inferring Errors in Systems Code," *Proc. ACM Symp. Operating Systems Principles*, pp. 57-72, 2001.

[33] U. Stern and D.L. Dill, "Automatic Verification of the SCI Cache Coherence Protocol," *Proc. Conf. Correct Hardware Design and Verification Methods*, pp. 21-34, 1995.

[34] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan, "Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pp. 258-269, 2002.

[35] R. Agrawal and R. Srikant, "Mining Sequential Patterns," *Proc. 11th Int'l Conf. Data Eng.*, 1995.

[36] X. Yan, J. Han, and R. Afshar, "CloSpan: Mining Closed Sequential Patterns in Large Datasets," *Proc. SIAM Int'l Conf. Data Mining*, May 2003.

[37] Z. Li, Z. Chen, S.M. Srinivasan, and Y. Zhou, "C-Miner: Mining Block Correlations in Storage Systems," *Proc. Third USENIX Conf. File and Storage Technologies*, pp. 173-186, 2004.

[38] A.V. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[39] R.E. Johnson and W.F. Opdyke, "Refactoring and Aggregation," *Proc. Int'l Symp. Object Technologies for Advanced Software*, pp. 264-278, 1993.

Zhenmin Li received the BE and ME degrees in computer science from Tsinghua University, China. He is a PhD student in the Department of Computer Science, University of Illinois at Urbana-Champaign. His research interests include computer systems, software reliability, data mining, storage systems, and energy management.

Shan Lu is a PhD student in the Department of Computer Science, University of Illinois at Urbana-Champaign. Her research interests include architectural and system support for software debugging and system configuration management.

Suvda Myagmar received the BS degree in computer science from Concord College and the MS degree in computer science from University of Illinois at Urbana-Champaign. She is a PhD student in the Department of Computer Science, University of Illinois at Urbana-Champaign. She is working on security issues of software defined radios. Her research interests include computer and network security, wireless communications, and reconfigurable platforms.

Yuanyuan Zhou received the MA and PhD degrees from Princeton University. She is currently an assistant professor in the Department of Computer Science, University of Illinois at Urbana-Champaign (UIUC). Prior to UIUC, she worked at NEC Research Institute as a scientist from 2000 to 2002. Her main research interests include database storage, architecture and OS support for software debugging, power management, and memory management. She is a member of the IEEE.