# CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code

Zhenmin Li, Shan Lu, Suvda Myagmar and Yuanyuan Zhou

*Department of Computer Science*
*University of Illinois at Urbana-Champaign, Urbana, IL 61801*

## ABSTRACT

Copy-pasted code is very common in large software because programmers prefer reusing code via copy-paste in order to reduce programming effort. Recent studies show that copy-paste is prone to introducing bugs and a significant portion of operating system bugs concentrate in copy-pasted code. Unfortunately, it is challenging to efficiently identify copy-pasted code in large software. Existing copy-paste detection tools are either not scalable to large software, or cannot handle small modifications in copy-pasted code. Furthermore, few tools are available to 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 including operating systems, and detects copy-paste related 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 28 copy-paste related bugs in the latest version of Linux and 23 in FreeBSD. In addition, we analyze some interesting characteristics of copy-paste in Linux and FreeBSD, including the distribution of copy-pasted code across different length, granularity, modules, degrees of modification, and various software versions.

## 1 Introduction

### 1.1 Motivation

Copying and pasting code is a common practice in software development. In order to reduce programming effort and shorten programming time, programmers prefer reusing a piece of code via copy-paste rather than rewriting similar code from scratch. Recent studies [6, 13, 25] have shown that a large portion of code is duplicated in software. For example, Kapser and Godfrey [25], using a copy-paste detection tool called CCFinder [24], found that 12% of the Linux file system code (279K lines) was involved in code cloning activity. Baker [6] found that in the complete source of the X Window system (714K lines), 19% of the code was identified as duplicates.

Using abstractions such as functions and macros to remove this duplication might improve software maintenance; however, much duplication will likely remain, for two possible reasons. First, some changes are usually necessary, and copy-paste is much easier and faster than abstraction. Another reason is that functions may impose higher overhead. However, the psychological reasons for large percentage of existing copy-pasted code are beyond the scope of this paper.

Copy-pasted code is prone to introducing errors. For example, Chou et al. [10] 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 28 copy-paste related bugs in the latest version of Linux and 23 in FreeBSD. Most of these bugs were previously unreported.

A major reason 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.

Figure 1 shows an example of a 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 by kernel developers [1]. 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, 14, 17, 32] or dynamic tools such as Purify [19], Valgrind [36], and CCured [12]. Besides this bug, CP-Miner has also detected many other similar bugs caused by copy-paste in Linux, FreeBSD, PostgreSQL and Web Apache.

While one can imagine augmenting the software development tools and editors with copy-paste tracking, this support does not currently exist. Therefore, we are fo-

```
( 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
118    }
        ......
143 }
```

Figure 1: An example of a copy-paste related error detected by CP-Miner. This bug appears in linux-2.6.6/arch/sparc64/prom/memory.c. A similar bug is also detected in file /arch/sparc/prom/memory.c.

cusing on detecting likely copied and pasted code in an existing code base. Not all code segments identified by previous detection tools and our tool are really the results of copy-paste (even though we prune many of the false copy-pasted segments as described in Section 3.1.4), but for simplicity we refer likely-copy-pasted segments as copy-pasted segments.

It is a challenging task to efficiently extract copy-pasted code in large software such as an operating system. Even though some previous studies [16, 20] have addressed the related problem of plagiarism detection, they are not suitable for detecting copy-pasted code. Those tools, such as the commonly used JPlag [33], were designed to measure the degree of similarity between a pair of programs in order to detect cheating. 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.

So far, only a few tools have been proposed to identify copy-pasted code in a single program. Examples of such tools include Moss [4, 35], Dup [6], CCFinder [24] and others [5, 7]. Most of these tools suffer from some or all of the following limitations:

(1) *Efficiency:* Most existing tools are not scalable to large software 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 [13, 22] can only detect copy-pasted segments that are exactly identical. Moreover, most of the existing tools do not allow statement insertions or modifications in a copy-pasted segment. 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 1-2 statements.

(3) *Bug detection:* The existing tools cannot detect copy-paste related bugs. They only aim at detecting copy-pasted code and do not consider bugs associated with copy-paste.

## 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 including operating system code, and also detects copy-paste related bugs. It requires no modification or annotation to the source code of software being analyzed. Our paper makes three main contributions:

**(1) A scalable copy-paste detection tool for large software:** CP-Miner can efficiently find copy-pasted code in large software including operating system code. Our experimental results show that it takes less than 20 minutes for CP-Miner to detect 150,000–190,000 different copy-pasted segments that account for about 20–22% of the source code in Linux and FreeBSD (each with more than 3 million lines of code). Additionally, it takes less than one minute to detect copy-pasted segments in Apache web server and PostgreSQL, accounting for about 17–22% of total source code.

Compared to CCFinder [24], CP-Miner is able to find 17–52% more copy-pasted segments 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 one shown in Figure 1, most of which are hard to detect with existing static or dynamic bug detection tools. More specifically, CP-Miner has detected 28 potential bugs in the latest version of Linux, 23 in FreeBSD, 5 in Web Apache, and 2 in PostgreSQL. Most of these bugs had never been reported.

We have reported these bugs to the corresponding developers. So far five bugs have recently been confirmed and fixed by Linux developers, and one bug has been confirmed and fixed by Apache developers.

**(3) Statistical study of copy-pasted code distribution in operating system code:** Few previous studies have been

[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 $1, 2, ..., \frac{n}{2}$, the total number of pairwise comparisons is $O(n^3)$.

conducted on the characteristics of copy-paste in large software. Our work analyzed some interesting statistics of copy-pasted code in Linux and FreeBSD. Our results indicate that (1) copy-pasted segments are usually not too large, most with 5–16 statements; (2) although more than 50% of copy-pasted segments have only two copies, a few (6.3–6.7%) copy-pasted segments are copied more than 8 times; (3) there is a significant number (11.3–13.5%) of copy-pasted segments at function granularity (copy-paste of an entire function); (4) most (65–67%) copy-pasted segments require renaming at least one identifier, and 23–27% of copy-pasted segments have inserted, modified, or deleted one statement; (5) different OS modules have very different copy-paste coverage: *drivers*, *arch*, and *crypt* have higher percentage of copy-paste than other modules in Linux; (6) 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 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 [6]; (2) *parse-tree-based*, in which pattern matching is performed on the parse-tree of the code to search for similar subtrees [7, 27]; (3) *token-based*, in which the program is divided into a stream of tokens and duplicate token sequences are identified [24, 33].

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 are not necessarily copy-pasted. This is because copy-paste is code-based rather than syntax-based, i.e., it reuses a piece of code rather than an abstract syntax structure.

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 software, or do not tolerate modifications made in copy-pasted code. In contrast, CP-Miner can address both challenges.

### 2.2 Frequent Subsequence Mining

CP-Miner is based on *frequent subsequence mining* (also called frequent sequence mining), an association analysis technique that discovers frequent subsequences in a sequence database [2]. Frequent subsequence mining is an active research topic in data mining [38, 39]. It has broad applications, including mining motifs in DNA sequences, analysis of customer shopping behavior, etc.

A subsequence is considered *frequent* when it occurs in at least a specified number of sequences (called *min_support*) in the sequence database. A subsequence is not necessarily contiguous in an original sequence. We denote the number of occurrences of a subsequence as its *support*. A sequence that contains a given subsequence is called a *supporting sequence* of this subsequence.

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 $\{a\colon 5, b\colon 4, c\colon 5, ab\colon 4, ac\colon 5, bc\colon 4, abc\colon 4\}$, where the numbers are the supports of the subsequences.

CP-Miner uses a recently proposed frequent subsequence mining algorithm called *CloSpan* (*Clo*sed *S*equential *P*atter*n* Mining)[38], which outperforms most previous algorithms. Instead of mining the complete set of frequent subsequences, CloSpan mines only the *closed subsequences*. A closed subsequence is the subsequence whose support is different from that of its supersequences. CloSpan mainly consists of two 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 non-closed 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.

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

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 $L_1$:

$$\begin{aligned} L_1 &= \{a, b, c\}; \\ L_2 &= \{ab, ac, bc\}; \\ L'_3 &= L_2 \times L_1 = \{abc, abb, abc, aca, acb, acc, bca, bcb, bcc\} \end{aligned}$$

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 with only the frequent items in its *suffix database*. A suffix database of a subsequence $s$ is the database of all the maximum suffixes of the sequences that contain $s$. In our example,

for the frequent sequence $ab$ in $L_2$, its suffix database is $D_{ab} = \{ced, cef, ch, ijc\}$, and only $c$ is a frequent item, so $ab$ is only concatenated with $c$ and we get a longer sequence $abc$ that belongs to $L'_3$.

The second idea for improving mining performance is to efficiently evaluate whether a concatenated subsequence is frequent. Rather than searching the whole database, CloSpan only checks certain suffixes. In our 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 $min\_sup$, $s$ 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.

Due to space limitation, a detailed discussion of the CloSpan algorithm can be found in [29, 38].

# 3 CP-Miner

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

## 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 and composes larger copy-pasted segments. For convenience of description, 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. Then, a basic block (i.e., a straight-line 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 such as identifier mapping, we can find basic copy-pasted segments, which can then 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 and types) 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, we have enhanced the basic frequent subsequence mining algorithm, CloSpan, to support gap constraints in frequent subsequences. This enhancement allows CP-Miner to tolerate 1–2 statement insertions, deletions, or modifications in 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 a statement inserted.

The main steps of the process to identify 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 are 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*: Various techniques including identifier mapping are used to prune false positives.

(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 combined one may not be copy-pasted, even though each smaller one is.

Like other copy-paste detection tools, CP-Miner can only detect copy-pasted segments, but cannot tell which segment is original and which is copy-pasted from the original. Fortunately, this limitation is not a big problem because in most cases it is enough for programmers to know what 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 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. The current prototype of the CP-Miner parser only works for programs written in C or C++, but it is easy to modify it for other programming languages.

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 are

mapped into the same token. Constants are handled in the same way as identifiers: constants of the same type are mapped into the same token. However, operators and keywords are handled differently, with each one mapped to a unique token. After all the components of a statement are tokenized, a hash value digest is computed using the "hashpjw" [3] hash function, chosen for its low collision rate. Figure 2 shows the hash value for each statement in the example shown in Figure 1 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.

After each statement is mapped, the program becomes a long number sequence. Unfortunately, the frequent subsequence mining algorithms need a collection of sequences (a sequence database) as described in 2.2, 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 the 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. 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 length. But this is not a problem for CloSpan because it can deal with sequences of different sizes. The example shown in Figures 1 and 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; (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.

| 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 } | |

Figure 2: An example of hashing statements

### 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. At 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 Figure 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 line 111–118 and line 92–99 are basic copy-pasted segments.

Unfortunately, the mining process is not as straightforward as expected. The main reason is that the original CloSpan algorithm was not designed exactly for our purpose, and nor were other frequent subsequence mining algorithms. Most existing algorithms including CloSpan have the following two limitations that we had to enhance 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 1 support for subsequence $abc$, even though $abc$ does not appear contiguously in $abdec$. It is possible to have a large gap in the occurrence of a frequent subsequence in one of its supporting sequences. Hence, its corresponding code segment would have several statements inserted. Such segment is unlikely to be copy-pasted.

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 max-

imum 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 insertion or deletions are allowed in copy-paste, whereas the gap constraint with $max\_gap = 1$ or $max\_gap = 2$ means that 1 or 2 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 their corresponding supporting sequences. To find copy-pasted code, we need to find the supporting sequences for each frequent subsequence.

We enhance 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 "non-expandable". CP-Miner repeats this process until all groups in the candidate set are non-expandable.

### *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) 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 of basic and composed copy-pasted segments. The pruning techniques include:

**(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 Figure 2, variable $prom\_phys\_total$ is changed into $prom\_prom\_taken$ (except the bug on line 117).

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 amount 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, if a variable $A$ from segment 1 is changed into $a$ in 75% of its occurrences in segment 2 but 25% of its occurrences is changed into other variables, the $ConflictRatio$ of mapping $A \rightarrow a$ is 25%. The $ConflictRatio$ for the whole mapping scheme between these two segments are 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 total identifier occurrences that are occurrences of $A$. If $ConflictRatio$ for two candidate copy-pasted segments 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%.

**(2) Pruning tiny segments:** Our mining algorithm may find tiny copy-pasted segments that consist of only 1-2 simple statements. If such a tiny segment cannot be combined with neighbors to compose a larger segment, it is removed from the copy-paste list. This is based on the observation that copy-pasted segments are usually not very small because programmers cannot save much effort in copy-pasting a simple tiny code segment.

CP-Miner uses the number of tokens to measure the size of a segment. 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:** If a pair of candidate copy-pasted segments overlap with each other, they are also considered false positives. CP-Miner stops extending the pair of copy-pasted segments once they overlap. For some program structures such as the *switch* statement that 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 non-expandable 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 (8) are found. We can only manually examine each identified copy-pasted segment because there are no traces that record programmers' copy-paste operations 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 all of the large basic blocks into small blocks with at most 30 statements before feeding to the mining algorithm. Therefore, the search tree is at most with depth 30. With this constraint of search tree, the mining complexity of CP-Miner is $O(n^2)$ in the worst case. Furthermore, the optimizations described in Section 2.2 make it more efficient in both time and space overheads than the worst case.

### 3.2 Detecting Copy-paste Related Bugs

As we have mentioned in Section 1, the main cause 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 itself 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.

| **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) |

Table 1: Identifier mapping in the example in Figure 1 (the number after each identifier indicates the number of occurrences).

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 based on this observation.

Therefore, the challenge is to decide when an inconsistency is likely to be a bug instead of a false positive of copy-paste. To address this challenge, we need to consider the programmers' intention. Our bug detection method is based on the following observation: if a programmer makes a change in a copy-pasted segment, the changed identifier is unlikely to be a bug. But if he/she changes an identifier in most places but forgets to change it in a few places, the unchanged identifier is 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 unlikely to be a bug because programmers *intentionally* change $A$ to other names. 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.

Based on the above observation, CP-Miner reexamines each non-expandable 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 $NumUnchanged$ means the number of occurrences that a given identifier is unchanged, and $NumTotal$ means the number of total occurrences of this identifier in a given copy-pasted segment. Therefore, the lower the $UnchangedRatio$, the more likely it is a bug, unless $UnchangedRatio = 0$, which means that all of its occurrences have been changed. 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 on 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. If $UnchangedRatio$ for an identifier is not zero

and not larger than the threshold, the unchanged places are reported as bugs. When CP-Miner reports a bug, the corresponding identifier mapping information is also provided to programmers to help in debugging. In the example shown on 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 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 possibility. It remains as our future work.

# 4 Methodology

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

| 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 |

Table 2: Software evaluated in our experiments.

We set the thresholds used in CP-Miner as following. 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. Without specifying, we use setting (2) by default.

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

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

In our experiments, we also compare CP-Miner with a recently proposed tool called *CCFinder* [24]. Similar to our tool, CCFinder also tokenizes identifiers, keywords, constant, operators, etc. But different 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% 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 run CP-Miner and CCFinder on an Intel Xeon 2.4GHz machine with 2GB memory.

# 5 Evaluation Results of CP-Miner

We first present the evaluation results of CP-Miner in this section, including the number of copy-pasted segments, the number of detected copy-paste related 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

**Detecting Copy-pasted Code** CP-Miner has found a significant number of copy-pasted segments in the evaluated software. In this software, copy-pasted code makes up 17.7–22.3% of the code base. Table 3 shows the numbers of copy-pasted segments and $CP\_Coverage$. 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 accounts for about 15% of the source code. We have manually examined 100 random pairs of copy-pasted segments from all potential copy-pasted segments in Linux (with $max\_gap$ = 1), and found a few (only 8) false positives. The large number of copy-pasted segments motivates a support in software development environments such as Microsoft Visual Studio to maintain copy-pasted code.

| Software | $max\_gap$ = 0 | | $max\_gap$ = 1 | |
|---|---|---|---|---|
| | #Segments | $CP\_Coverage$ | #Segments | $CP\_Coverage$ |
| 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% |

Table 3: The number of copy-pasted segments and $CP\_Coverage$

Our results also show that a large percentage (30–50%) of copy-pasted segments have statement insertions and modifications. For example, when $max\_gap$ is 1, CP-Miner finds 62.4% more copy-pasted segments in Linux. In FreeBSD, the $CP\_Coverage$ increases from 14.9% to 20.4% when $max\_gap$ is relaxed from 0 to 1. These results show that previous tools including CCFinder that 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 limitation, 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.

**Detecting Copy-paste Related Bugs** CP-Miner has also reported many copy-paste related errors in the evaluated software. Since the errors reported by CP-Miner may not be bugs, we verify each reported error manually and then report to the corresponding developer community those errors that we suspect to be bugs with high confidence. The

| Software | errors reported | bugs verified | careless programming | false alarms | | |
|---|---|---|---|---|---|---|
| | | | | (1) | (2) | (3) |
| Linux | 421 | 28 | 21 | 151 | 41 | 57 |
| FreeBSD | 443 | 23 | 8 | 307 | 41 | 30 |
| Apache | 17 | 5 | 0 | 3 | 1 | 6 |
| PostgreSQL | 74 | 2 | 0 | 13 | 10 | 43 |

Table 4: Errors reported by CP-Miner ($UnchangedRatio$ threshold = 0.4) and bugs verified by us with high confidence, some of which are confirmed and fixed by corresponding developers after we reported. 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 immediate future work.

numbers of errors found by CP-Miner and verified bugs are shown on Table 4. The results are achieved by setting the $UnchangedRatio$ threshold to be 0.4.

Both Linux and FreeBSD have many copy-paste related bugs. So far, we have verified 28 and 23 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. Recently, five Linux bugs have been confirmed and fixed by 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 much fewer copy-paste related bugs. We have verified 5 bugs for Apache and 2 bugs for PostgreSQL with high confidence. One bug in Apache was immediately fixed by the Apache developers after we reported it to them.

In addition to those bugs verified, we also find many "potential bugs" (21 in Linux and 8 in FreeBSD) that are not bugs by coincidence but might become bugs in the future. We call this type of errors "careless programming". Similar to the bugs verified, these errors also forget to change some identifiers consistently at a few places. Fortunately, by coincidence, the new identifiers and the old ones happen to have the same values. However, if such implicit assumptions are violated in future versions of the software, it would lead to bugs that are hard to detect.

## 5.2 False Alarms

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

**(1) Incorrectly matched copy-pasted segments:** In some copy-pasted segments that contain multiple "*case*" or "*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 composition might be chosen. As a result, identifiers are compared between two incorrectly matched copy-pasted segments, which results in false alarms.

These false alarms 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 of such false alarms can be pruned.

**(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 alarms are caused by such exchangeable orders.

These false alarms 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 software 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 million 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 and FreeBSD. For Apache and PostgreSQL, CP-Miner consumes 27–57 MB of memory.

| 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 |

Table 5: Execution time and memory space of CP-Miner

## 5.4 Comparison with CCFinder

We have compared CP-Miner with CCFinder [24]. CCFinder has execution time similar to that of CP-Miner, but CP-Miner discovers much more copy-pasted segments. In addition, CCFinder does not detect copy-paste related bugs. As we explained in Section 4, CCFinder allows identifier-renaming but not statement insertions. In addition, pruning in CCFinder is not so rigorous as CP-Miner. For example, CCFinder reports incomplete statements in copy-pasted segments, which is unlikely in practice. After pruning the incomplete statements, many small copy-

| Software | CCFinder | CP-Miner |
|---|---|---|
| Linux | 14.7%(19.8%) | 22.3% |
| FreeBSD | 14.5%(19.6%) | 20.4% |
| Apache | 11.8%(15.3%) | 17.7% |
| PostgreSQL | 18.5%(23.8%) | 22.2% |

Table 6: $CP\_Coverage$ comparison between CP-Miner and CCFinder. For CCFinder, the first number is the result after pruning those incomplete, small segments, and the second number in parentheses is the result before pruning.

pasted segments consist of less than 30 tokens, which are too simple to be worth copying.

CP-Miner can identify 17–52% more copy-pasted code than CCFinder because CP-Miner can tolerate statement insertions and modifications. Table 6 compares the $CP\_Coverage$ 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 incomplete, small segments from CCFinder's output.

### 5.5 Effects of Threshold Settings

**Segment Size Threshold** Figure 3 shows the effect of segment size threshold $min\_size$ on $CP\_Coverage$. As expected, $CP\_Coverage$ decreases when $min\_size$ increases because more copy-pasted segments are pruned. The results also show that the decrement slowdowns 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.

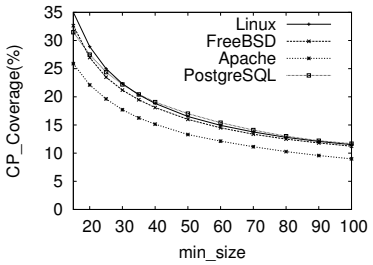



Figure 3: Effects of $min\_size$ on $CP\_Coverage$

**Unchanged Ratio Threshold** Figure 4 shows the effect of unchanged ratio threshold on the number of bugs reported. Since $UnchangedRatio \geq 0.5$ means that most of the identifiers are not changed after copy-pasting, 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 $UnchangedRatio$ threshold less than 0.5.

As expected, more errors are reported by CP-Miner when the $UnchangedRatio$ threshold increases. Specifically, the number of errors reported increases gradually

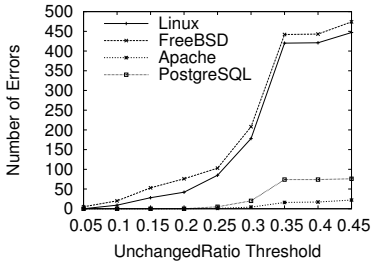
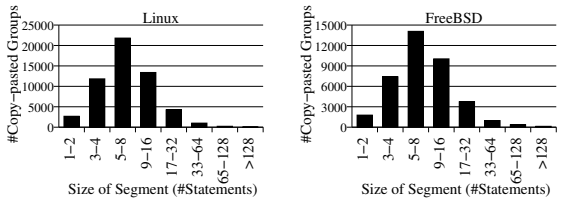
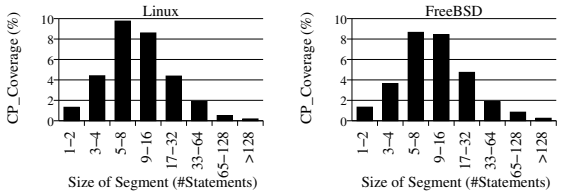



Figure 4: Effects of $UnchangedRatio$ threshold on errors reported




(a) # of copy-paste groups with various segment sizes (# of statements)




(b) The $CP\_Coverage$ with various segment sizes (# of statements)

Figure 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 $CP\_Coverage$ does not equal to the overall $CP\_Coverage$.

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 alarms during our verification. For example, CP-Miner reports many errors where only 1 out of 3 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 alarms, we can further analyze the identifiers in the context of the copy-pasted segments (e.g., the whole function). We leave this improvement as our future work.

## 6 Statistics of Copy-paste in OS code

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

### 6.1 Copy-paste Size and Granularity

Figure 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%) copy-pasted segments are not very large, with only 5–16 statements. Only a few (0.2–5.0%) copy-pasted segments have more than 64 statements. In particular, Figure 5(a) shows that most (35–40%) copy-paste groups contain 5–8 statements in each segment. Figure 5 (b) shows similar characteristics: copy-pasted segments with 5–8 statements cover about 7–10% of the source code.

Figure 6 shows the distribution of copy-paste group size. About 60% 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, a lot of code is replicated more than once.

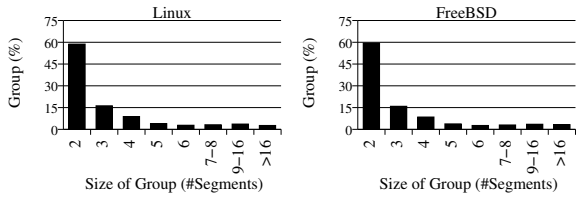



Figure 6: Copy-paste group size distribution in terms of the number of segments in each group. Each bar represents the percentage of copy-paste groups that contain the corresponding number of segments.

| Software | basic block | function |
|---|---|---|
| Linux | 17,818 (9.0%) | 26,744 (13.5%) |
| FreeBSD | 13,999 (9.1%) | 17,254 (11.3%) |

Table 7: Distribution of copy-paste granularity: numbers and percentages of copy-pasted segments at different granularity. Note here the percentage is not $CP\_Coverage$. It is calculated by comparing to the total number of copy-pasted segments.

Total 6.3–6.7% of copy-pasted segments are copy-pasted more than 8 times. If a bug is detected in one of the copies, it is difficult for programmers to remember fixing the bug in the other 8 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 9% of copy-pasted segments are basic blocks, which indicates that programmers seldom copy-paste basic blocks because most of them are too simple to worth it.

More interestingly, there are 13.5% of copy-pasted segments with whole functions in Linux and 11.3% in FreeBSD. The reason is that many functions provide similar functionalities, such as reading data from different types of devices. Those functions can be copy-pasted with modifications such as replacing data types of parameters. This motivates some refactoring tools [23] to better maintain these copy-pasted functions.

## 6.2 Modifications in Copy-pasted Segments

Figure 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 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 65–67% of copy-pasted segments require identifier renaming. For example, in Linux, 27% copy-pasted segments are identical, and 8% segments are almost identical with only one statement inserted. The rest 65% 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.

The results in Figure 7 also show that about 23–27% of copy-pasted segments contain at least one statement insertion, deletion, and modification ($Gap$=1). It indicates that it is important for copy-paste detection tools to tolerate such statement modifications.

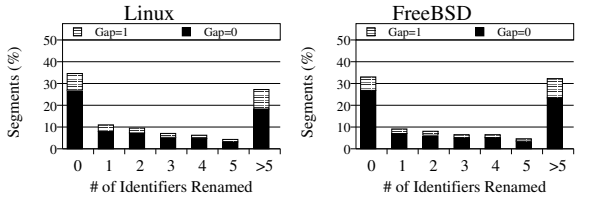
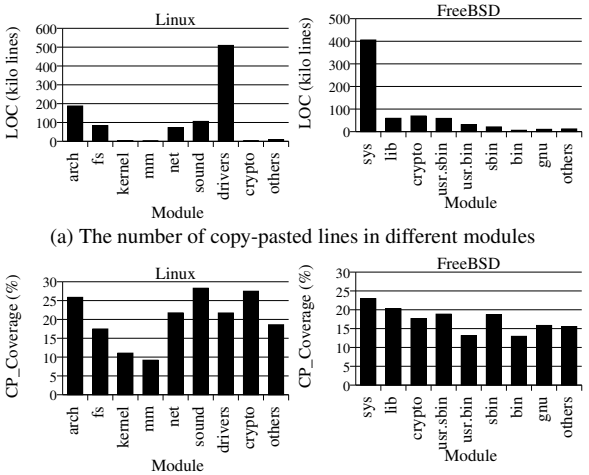



Figure 7: Distribution of identifiers changed in copy-pasted segments. Each bar represents the percentage of segments that have the corresponding number of renamed identifiers. Each bar has two parts: "$Gap = 0$" and "$Gap = 1$" represent the copy-pasted segments with no and one statement modifications,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




(a) The number of copy-pasted lines in different modules

(b) $CP\_Coverage$ in different modules

Figure 8: Copy-pasted code in different modules.

## 6.3 Copy-pasted Code across Modules

Different modules have different copy-paste characteristics. In this subsection, we analyze copy-pasted code across different modules in operating system code. We split Linux into 9 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 9 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.

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

Figure 8 (a) 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% of all copy-pasted code in Linux, and module "sys" accounts for 60% 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.

Figure 8 (b) shows that a large percentage (20–28%) of

the code in Linux is copy-pasted in the "arch" module, the "crypto" module, 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 sub-modules 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.

### 6.4 Evolution of Copy-paste

Figure 9 shows that the copy-pasted code increases as the operating system code evolves. For example, Figure 9(a) shows that 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 through version 1.0 to 2.6.6.

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

## 7 Related Work

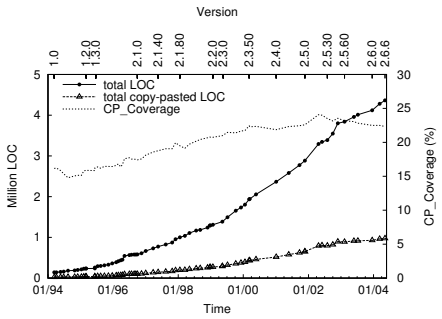
In this section, we briefly discuss closely related work that has not been described in earlier sections.

### 7.1 Detecting Copy-Pasted Code

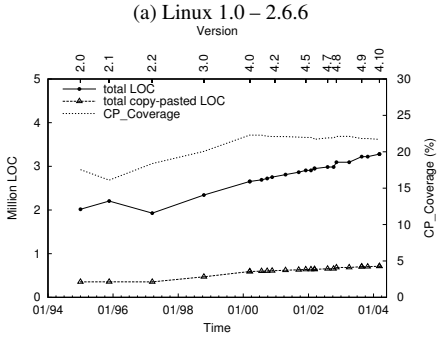
Several studies have been conducted on detection of copy-pasted code. The techniques used include: line-by-line [6], token-by-token [24, 33], fingerprinting [21], visualization [11, 13], abstract syntax tree [7, 27], and dependence graph [26, 28].

*Dup* [6] 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 like the presence or absence of new lines. In addition, it does not find non-contiguous copy-pastes. CP-Miner does not have these shortcomings.

Johnson [21] proposed using a fingerprinting algorithm on a substring of the source code. In this algorithm, calculated signatures 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.




(a) Linux 1.0 – 2.6.6




(b) FreeBSD 2.0 – 4.10

Figure 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 through 1.0 to the current version 2.6.6. The versions of FreeBSD include the main branch through 2.0 to 4.10.

Some graphical tools were proposed to understand code similarities in different programs (or in the same program) visually. *Dotplots* [11] of source code can be constructed by tokenizing the code into lines and placing a dot in coordinates $(i, j)$ on a 2-D graph, if the $i^{th}$ input token matches $j^{th}$ input token. Similarly, *Duploc* [13] 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.

Baxter et al. [7] proposed a tool that transforms source code into abstract-syntax trees (AST), and detects copy-paste by finding identical subtrees. Similar to other tools, it is not tolerant to modifications in copy-pasted segments. In addition, it may introduce many false positives because two code segments with the same syntax subtrees are not necessarily copy-pastes.

Komondoor et al. [26] proposed using 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 from

*bison*, much slower than CP-Miner. Another slow PDG-based approach is found in [28].

Mayrand et al. [31] 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* [33], *Moss* [35], and *sif* [30] are tools to find similar programs among a given set. They have been commonly used to detect plagiarism. Most of them are not suitable for detecting copy-pasted code in a single large program.

Kontogiannis et al. [27] 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.

### 7.2 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 [19], Valgrind [36], DIDUCE [18], Eraser [34], and CCured [12]. Dynamic tools have more accurate information but may introduce 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 [15, 32, 37] and program analysis [8, 14, 17]. Most static tools require significant involvement of programmers to write specifications or annotate programs. But the advantage of static tools is that they add no overhead during execution, and it can find bugs that may not occur in the common execution paths. A few tools do not require annotations, but they focus on detecting different 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 any annotation requirement from 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 those semantic ones, cannot be found by previous tools.

Our work is motivated by and related to Engler et al.'s empirical analysis of operating systems errors [10]. Their study gave an overall error distribution and evolution analysis in operating systems, and found that copy-paste is one of the major causes for bugs. Our work presents a tool to detect copy-pasted code and related bugs in large software including operating system code. Many of these bugs such as the one in Figure 1 cannot be detected by their tools.

## 8 Conclusions

This paper presents a tool called CP-Miner [2] that uses data mining techniques to efficiently identify copy-pasted code in large software including operating systems, and also 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% of the source code in Linux and FreeBSD. Moreover, CP-Miner has detected 28 and 23 copy-paste related bugs in the latest versions of Linux and FreeBSD, respectively. Compared to CCFinder [24], CP-Miner finds 17–52% 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 across different segment sizes, group sizes, granularity, modules, amount of modifications, and software evolution.

Our results indicate that maintaining copy-pasted code would be very useful for programmers because it is commonly used in large software such as operating system code, and it can easily introduce hard-to-detect bugs. We hope our study motivates software development environments such as Microsoft Visual Studio 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 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 it relies on semantic information. It is impossible to guess what the programmer would want to insert or modify. Another type of copy-paste related bugs is caused by programmers forgetting to fix a known bug in all copy-pasted segments. They only fix one or two segments but forget to change it in the others. Our tool CP-Miner can detect simple cases of this type of errors. But if the fix is too complicated, CP-Miner would miss the bug because the modified code segment becomes too different from the others to be identified as copy-paste. To solve this problem more thoroughly, it would require support from software development environments such as Microsoft Visual Studio.

## 9 Acknowledgements

The authors would like to thank the shepherd, Andrew Myers, the anonymous reviewers, and James Larus (Microsoft Research Lab) for their invaluable feedback. We appreciate Professor Jiawei Han and his group for their CloSpan mining algorithm. We would also like to thank Cigdem Sengul for her help with the initial investigation of our project. This research is supported by IBM Faculty

[2] CP-Miner will be released to the research community.

Award, NSF CNS-0347854 (career award), NSF CCR-0305854 grant and NSF CCR-0325603 grant. Our experiments were conducted on equipment provided through the IBM SUR grant.

## REFERENCES


[1] Linux kernel mailing list. http://lkml.org.

[2] R. Agrawal and R. Srikant. Mining sequential patterns. In *Eleventh International Conference on Data Engineering*, 1995.

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

[4] A. Aiken. Moss: A system for detecting software plagiarism. *http://www.cs.berkeley.edu/˜aiken/moss.html*.

[5] B. S. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, 24:49–57, 1992.

[6] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of the Second Working Conference on Reverse Engineering*, page 86. IEEE Computer Society, 1995.

[7] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, page 368. IEEE Computer Society, 1998.

[8] 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. In *Proceeding of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 258–269. ACM Press, 2002.

[9] A. Chou, B. Chelf, D. R. Engler, and M. Heinrich. Using meta-level compilation to check FLASH protocol code. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating System*, pages 59–70. ACM Press, 2000.

[10] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An empirical study of operating system errors. In *Symposium on Operating Systems Principles*, pages 73–88, 2001.

[11] K. W. Church and J. I. Helfman. Dotplot: A program for exploring self-similarity in millions of lines of text and code. *Journal of Computational and Graphical Statistics*, 1993.

[12] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. CCured in the real world. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 232–244. ACM Press, 2003.

[13] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of International Conference on Software Maintenance*, pages 109–118. IEEE, 1999.

[14] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 237–252. ACM Press, 2003.

[15] D. Engler, D. Y. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 57–72. ACM Press, 2001.

[16] S. Grier. A tool that detects plagiarism in Pascal programs. In *Proceedings of the 12th SIGCSE Technical Symposium on Computer Science Education*, pages 15–20. ACM Press, 1981.

[17] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 69–82. ACM Press, 2002.

[18] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the International Conference on Software Engineering*, May 2002.

[19] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, pages 158 – 185, Dec 1992.

[20] H. T. Jankowitz. Detecting plagiarism in student Pascal programs. *Computer Journal*, 31(1):1–8, 1988.

[21] J. H. Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of the conference of the Centre for Advanced Studies on Collaborative research*, Toronto, Ontario, Canada, October 1993.

[22] J. H. Johnson. Substring matching for clone detection and change tracking. In *Proceedings of the International Conference on Software Maintenance*, pages 120–126. IEEE Computer Society, 1994.

[23] R. E. Johnson and W. F. Opdyke. Refactoring and aggregation. In *Object Technologies for Advanced Software, First JSSST International Symposium*, volume 742, pages 264–278. Springer-Verlag, 1993.

[24] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[25] C. Kapser and M. W. Godfrey. Toward a taxonomy of clones in source code: A case study. *Evolution of Large-scale Industrial Software Applications (ELISA)*, Sept 2003.

[26] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *8th International Symposium on Static Analysis (SAS)*, 2001.

[27] K. Kontogiannis, M. Galler, and R. DeMori. Detecting code similarity using patterns. *Working Notes of the Third Workshop on AI and Software Engineering: Breaking the Toy Mold (AISE)*, 1995.

[28] J. Krinke. Identifying similar code with program dependence graphs. In *Eighth Working Conference on Reverse Engineering (WCRE)*, 2001.

[29] Z. Li, Z. Chen, S. Srinivasan, and Y. Zhou. C-Miner: Mining Block Correlations in Storage Systems. In *Proceedings of the 3rd USENIX Conference on File and Storage Technology*, 2004.

[30] U. Manber. Finding similar files in a large file system. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 1–10, San Fransisco, CA, USA, 17–21 1994.

[31] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the 1996 International Conference on Software Maintenance*, page 244. IEEE Computer Society, 1996.

[32] M. Musuvathi, D. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, Dec. 2002.

[33] L. Prechelt, G. Malpohl, and M. Philippsen. Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science*, 8(11):1016–1038, Nov 2002.

[34] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[35] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 76–85. ACM Press, 2003.

[36] J. Seward. Valgrind, an open-source memory debugger for x86-GNU/Linux. available at URL http://developer.kde.org/ sewardj/.

[37] U. Stern and D. L. Dill. Automatic verification of the SCI cache coherence protocol. In *Conference on Correct Hardware Design and Verification Methods*, pages 21–34, 1995.

[38] X. Yan, J. Han, and R. Afshar. CloSpan: Mining closed sequential patterns in large datasets. In *Proceedings of 2003 SIAM International Conference on Data Mining (SDM'03)*, San Fransisco, CA, May 2003.

[39] M. Zaki. SPADE: An efficient algorithm for mining frequent sequences. *Machine Learning*, 40:31–60, 2001.