CBCD: Cloned Buggy Code Detector

Jingyue Li
DNV Research & Innovation
Høvik, Norway
Jingyue.Li@dnv.com

Michael D. Ernst
University of Washington
Seattle, WA, USA
mernst@uw.edu

***Abstract*—Developers often copy, or clone, code in order to reuse or modify functionality. When they do so, they also clone any bugs in the original code. Or, different developers may independently make the same mistake. As one example of a bug, multiple products in a product line may use a component in a similar wrong way. This paper makes two contributions. First, it presents an empirical study of cloned buggy code. In a large industrial product line, about 4% of the bugs are duplicated across more than one product or file. In three open source projects (the Linux kernel, the Git version control system, and the PostgreSQL database) we found 282, 33, and 33 duplicated bugs, respectively. Second, this paper presents a tool, CBCD, that searches for code that is semantically identical to given buggy code. CBCD tests graph isomorphism over the Program Dependency Graph (PDG) representation and uses four optimizations. We evaluated CBCD by searching for known clones of buggy code segments in the three projects and compared the results with text-based, token-based, and AST-based code clone detectors, namely Simian, CCFinder, Deckard, and CloneDR. The evaluation shows that CBCD is fast when searching for possible clones of the buggy code in a large system, and it is more precise for this purpose than the other code clone detectors.**

***Keywords*- Validation, Debugging aids**

I. INTRODUCTION

Although copy-paste is generally regarded as a bad coding practice, it is sometimes necessary, and some developers do it to save development effort. Baker found that 24% of files examined included exact matches of code lines [4]. Ducasse et al. reported that two files of gcc have more than 60% duplication [3]. A study of code clones in Linux [2] showed that:

- A few copy-pasted segments were copied more than eight times.
- Device drivers and cryptography have the highest percentage of clones, because many drivers share similar functionality and cryptographic algorithms consist of multiple similar computational steps.

Code copy-paste and software reuse makes buggy code appear in multiple places in a system or in different systems. For example, code clones and software reuse have caused duplicated software security vulnerabilities [18]. Cut-and-paste is a major cause of operating system bugs [11].

This paper makes two contributions. First, we examined the data in the SCM (Software Configuration Management System) of 4 projects: an industrial software product line, the Linux kernel, Git, and PostgreSQL. We discovered that identical buggy code does exist in all 4 projects.

Second, to find clones of buggy code, we developed a clone detection tool, CBCD. Given an example of buggy code, CBCD uses isomorphism matching in the Program Dependence Graph (PDG) [15] to search for identical code — that is, clones. Subgraph isomorphism is NP-complete [13], so we implemented four optimizations that reduce the number and complexity of graphs in the PDG isomorphism matching. Evaluation of CBCD on real cloned buggy code confirms that CBCD is scalable to large systems. To evaluate how well CBCD can find cloned bugs, we also compared CBCD with text-based, token-based, and AST-based code clone detectors, using the identified buggy codes and their clones as oracles. CBCD outperformed the other approaches. (Our evaluation focuses on the important problem of finding clones of buggy code. For other tasks, the other clone detectors may be better than CBCD.)

The rest of this paper is organized as follows. Section 2 presents our empirical study of cloned buggy code in one commercial product line and three large open source systems. Section 3 describes the design and implementation of CBCD, which can find cloned buggy code. Section 4 presents our experimental evaluation. Section 5 discusses related work, and Section 6 concludes.

II. AN EMPIRICAL STUDY OF CLONED BUGGY CODE

We first manually investigated whether buggy lines of code are cloned in real systems. We examined the SCM of the Linux kernel, Git, and PostgreSQL, and the bug reporting system of a commercial software product line.

A. *The Linux Kernel*

For the Linux kernel, we searched for the keywords in Table I in commit messages and in the bug tracking system, which records discussions between developers during debugging. For each match, we read the description of the commit, the discussions between developers, and the “diff” of the original file and the changed file. This information indicated to us whether the commit was necessitated by duplication of a bug. If so, we identified the buggy code and its clones manually.

The second column of Table I shows the number of distinct, independent bugs that exist in multiple locations. By distinct, we mean that we count a bug once, even if it appears in 3 places. By independent, we mean that if a commit message said, “The same problem as commit #1234”, we count only one of the two bugs. Finally, there is no double-counting: if a commit message said “the same problem as #1234, with the same fix", then it only appears in one row of Table I. Some examples of these cloned bugs are shown in Table II. However, for some of these bugs, we cannot locate the cloned buggy code, because the developers did not give enough details. The third column of Table I omits such bugs. For example, one developer said, "The same bug that existed in the 64bit memcpy() also exists here so fix it here too" but did not specify which version of which file of the system includes the fix of the bug in 64bit memcpy(). As there are many files and many versions of Linux, it would be difficult to search all of them to find the fixes to memcpy(). Even if we found a change to memcpy(), without further information, we do not know if that change is the fix mentioned by the developer.

TABLE I. CLONED BUGS WHICH EXIST IN MORE THAN ONE PLACE IN THE LINUX KERNEL

Key words used for searching the SCM	Number of distinct bugs existing in more than one place	Number of bugs whose clones we can locate
same bug	53	23
same fix	48	24
same issue	62	39
same error	7	6
same problem	112	65
Sum	282	157

TABLE II. EXAMPLES OF CLONED BUGS IN THE LINUX KERNEL

Phrases in the SCM explaining the cloned bugs	Code modified (i.e., the lines of code modified by the bug fix)
This is quite the same fix as in 2cb96f86628d6e97fcbda5f e4d8d74876239834c	static int my_atoi(const char *name){ int val = 0; for (;; name++) { switch (*name) { case '0' ... '9': val = 10*val+(*name-'0'); break; default: return val;} }}
This patch fixes iwl3945 deadlock during suspend by moving notify_mac out of iwl3945 mutex. This is a portion of the same fix for iwlwifi by Tomas.	ieee80211_notify_mac(priv->hw, IEEE80211_NOTIFY_RE_ASSOC);
It turns out that at least one of the caller had the same bug.	ret = btrfs_drop_extents(trans, root, inode, start, aligned_end, start, &hint_byte);
Other platforms have this same bug, in one form or another	atomic_inc(&call_data->finished); func(info);

B. Git and PostgreSQL

For the Git and PostgreSQL projects, we used the same methodology. Table III shows the number of bugs that exist in multiple places.

C. A Commercial Software Product Line

We also evaluated a commercial product line in which a single product is produced for more than 40 different operating systems and mobile devices. For 17 of the projects, we have access to bug reports and developer discussions. These projects have a total of 25420 valid bugs that are confirmed and resolved as a bug in the code, not a user error.

We searched for the same keywords in the bug reports. Unlike the Linux kernel, Git, and PostgreSQL, we do not have full access to the source code in the SCM. Thus, we did not check the code differences. Our assessment of whether a bug was duplicated (as shown in Table IV) was based on reading the discussions between developers during debugging. It turns out that 3.8% (969/25420) of the bugs in these 17 projects exist in more than one place.

TABLE III. CLONED BUGS WHICH EXIST IN MORE THAN ONE PLACE IN GIT AND POSTGRESQL

Key words used for searching the SCM	GIT		POSTGRESQL	
	Number of distinct bugs existing in more than one place	*Number of bugs whose clones we can locate*	*Number of distinct bugs existing in more than one place*	*Number of bugs whose clones we can locate*
same bug	7	5	9	9
same fix	7	4	5	4
same issue	14	3	2	0
same error	0	0	1	8
same problem	5	0	16	1
Sum	33	12	33	22

TABLE IV. CLONED BUGS WHICH EXIST IN MORE THAN ONE PLACE IN THE COMMERCIAL SOFTWARE PRODUCT LINE

Key words used for searching the bug reports	Number of distinct bugs existing in more than one place
same bug	170
same fix	40
same issue	302
same error	56
same problem	401
Sum	969

III. CBCD, A TOOL TO SEARCH FOR CLONED BUGGY CODE

Once a bug is detected, it is necessary to check the whole system to see if the bug exists somewhere else. Section II shows that this is not merely a theoretical concern, but is important in practice. It is especially important for a software product line, because of high similarity among products. Customer satisfaction drops when a customer re-encounters a bug that the vendor claimed to have fixed. Although regression testing can check whether a bug is fixed, or can detect an identical manifestation of the bug in other products, regression testing cannot find all occurrences of the bug, especially when testers do not know where the buggy code may appear. Thus, it is important to supplement regression testing by a search for clones to locate code that may behave similarly to the buggy code.

D. PDG Based Code Clone Detectors

Some buggy lines may be copy-pasted "as-is", but often, developers slightly modify the copy-pasted code to fit a new context [2]. More than 65% of copy-pasted segments in Linux require renaming at least one identifier, and code insertion and deletion happened in more than 23% of the copy-pasted segments [2]. Statement reordering, identifier renaming, and statement insertion or deletion are also common in buggy code clones, especially clones introduced due to code or component reuse. For example, in Table II, a developer stated that "Other platforms have this same bug, in one form or another."

Our approach is to adapt Program Dependence Graph (PDG)-based code clone detection methods [7, 8, 9, 10], because we believe that the PDG-based approach is more resilient to code changes than text-based, token-based, and AST-based approaches.

E. Tool Architecture

Our tool, CBCD (for "Cloned Buggy Code Detector") has a pipe-and-filter architecture, as shown in Fig. 1. CBCD represents a program or code fragment as a PDG, which is a directed graph. Each vertex represents an entity of the code, such as a variable, statement, and so on; CBCD also records the vertex kind (e.g., "control-point", "declaration", or "expression"), the position (i.e., the file name and the line of the represented source code), and the source code text itself. Each edge of a PDG represents control or data dependency between two vertexes.

CBCD's algorithm consists of three steps.

Step 1: CodeSurfer [14] generates the PDG of both the buggy code (the "Bug PDG") and of the system to be searched for clones of the buggy code (the "System PDG"). The Bug PDG may consist of multiple sub-graphs depending on the structure of the buggy code; CBCD handles this case, but for simplicity of presentation this paper assumes the Bug PDG is connected. The System PDG consists of a collection of interlinked per-procedure PDGs.

Step 2: CBCD prunes and splits the System PDG (see Section III.C) to reduce its complexity and make subgraph checking cheaper. Optionally, CBCD also splits the original Bug PDG into multiple smaller PDGs (see Section III.C.4).

Step 3: CBCD determines whether the Bug PDG is a subgraph of the System PDG. It uses igraph's [16] implementation of subgraph isomorphism matching. igraph is faster than other tools, such as Nauty [17], when comparing randomly-connected graphs with less than 200 nodes [12].

CBCD filters the matches reported by igraph. CBCD only outputs matches where, for each corresponding vertex, the vertex kinds match and the source code text matches. When comparing vertex kinds, CBCD tolerates control replacement, e.g., when developers change a "for" loop to a "while" loop to provide the same functionality. When comparing source code text, vertexes that represent parameters of a function call are exempted. Note that even if all vertex kinds and text match identically (which CBCD does not require), the source code could still be different so long as it led to the same PDG. For example, reordering of (non-dependent) statements does not affect the PDG, nor does insertion of extra statements, such as debugging printf statements.

CBCD aims to find all semantically identical code clones. Two code snippets are semantically identical if there is no program context that can distinguish them—that is, if one snippet is substituted for the other in a program, the program behaves identically to before, for all inputs. Determining semantic equivalence is undecidable, so CBCD reports code with matching PDGs. As a result, every match that CBCD finds is semantically identical to the buggy code, but CBCD is not guaranteed to find all semantically-identical clones.

F. Pruning the Search Space for Isomorphism Graph Matching

All code clone detection tools that rely on graph matching face scalability problems. CBCD's isomorphism matching step is the most time-consuming step, especially for matching two big graphs. The reason for this is that subgraph isomorphism identification is NP-complete [13]. In the worst case, the fast subgraph isomorphism algorithm [12] implemented by igraph [16] requires O(N!N) time, where N is the sum of the number of nodes and edges of both graphs to be compared. Liu et al. [9] claim that "PDGs cannot be arbitrarily large as procedures are designed to be of reasonable size for developers to manage." In practice, a procedure can be very big. For example, we used Git as a subject program, and its "*handle_revision_opt*" procedure has 817 vertexes and 2479 edges. But, even smaller comparisons can be intractable in practice. Consider a modest example: the buggy code has 5 lines of code (with around 10 vertexes and 15 edges in the PDG) and the procedure has 100 lines of code (around 200 vertexes and 300 edges). In this example, N = 525 and N!N is 3.6×10^{1204}.

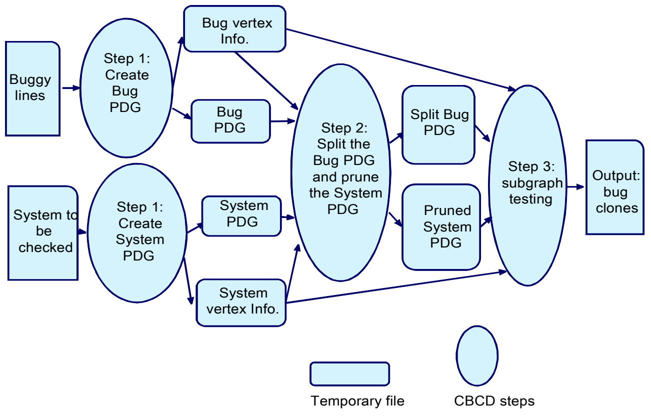

Figure 1. Architecture of CBCD

To deal with the scalability problem, Step 2 of CBCD prunes the number and complexity of the graphs to be compared.

We have implemented four optimizations. The first three optimizations are *sound*: each never excludes a true match, but makes the algorithm faster overall. These optimizations are run by default. The fourth optimization runs only if the buggy code segment contains too many lines of code.

The first three optimizations are based on the fact, explained in Section III.B, that CBCD reports system code as a clone of buggy code only if both the shape of the respective PDGs, and also the vertex kind and source text of corresponding vertices, are identical. The first three optimizations can be viewed as enhancements to the subgraph isomorphism checker, working around its limitation that it does not account for vertex kinds and source text.

All four optimizations are also based on the following observation: In most cases, the Bug PDG is small. Fig. 2

validates this observation: it is the maximum number of contiguous lines of code in each of the 163 Git, Linux kernel, and PostgreSQL bugs for which we can locate their cloned bugs. (This excludes 28 bug fixes that added code rather than changing code.) More than 88% of the bugs cover 4 or fewer contiguous lines of code.

1) Optimization 1 (Opt1): Exclude Irrelevant Edges and Nodes from the System PDG

CBCD removes every edge that cannot match an edge in the Bug PDG, because such an edge is irrelevant for CBCD's purposes. In particular, CBCD removes every edge whose start and end vertex kinds and vertex text are not included in the start and end vertex kinds and characters of an edge in the Bug PDG. In the best case, this disconnects entire sets of nodes, but it is useful even if it merely removes edges, because a single System PDG can be very big.

For example, suppose the Bug PDG has two edges: one from vertex kind "control-point" to vertex kind "expression", and the other from "expression" to "actual-in". Then, CBCD excludes from the System PDG all edges that do not start with "control-point" and end with "expression", or start with "expression" and end with "actual-in".

At this point, CBCD also compares the vertex characters (source code text), for vertex kinds whose code must match (e.g., not procedure parameters nor arguments). CBCD discards those with text that cannot match the Bug PDG. The purpose of comparing vertex kinds and characters is different than Step 3 of Section III.B. The comparison here excludes System PDG vertexes and edges that are irrelevant to the Bug PDG. The comparison in Step 3 ensures that the vertexes in the isomorphism matching graphs are also identical.

2) Optimization 2 (Opt2): Break the System PDG into Small Graphs

This optimization transforms the System PDG from one large graph into multiple small ones. CBCD must run more subgraph isomorphism matchings, but each matching will focus on a smaller graph. The idea is to utilize the vertex kind information of the Bug PDG to choose only small sections of the procedure PDG for each subgraph isomorphism matching. The steps of Opt2 are:

- **Opt2-step1:** Count the number of nodes of each vertex kind in the Bug PDG and the System PDG.
- **Opt2-step2:** Choose the vertex kind vk_{min} in the Bug PDG that has the minimum number of occurrences in the System PDG. If it occurs 0 times in the System PDG, there is no graph match.
- **Opt2-step3:** Calculate the pseudo-radius d_b of the Bug PDG: the greatest distance between a node of vertex kind vk_{min} and any other node.
- **Opt2-step4:** For each node of vertex kind vk_{min} in the System PDG, find the neighbor graph of the vertex, with radius d_b from the node of kind vk_{min}.

The distance computations ignore edge directions.

Fig. 3 shows an example. Since the nodes of vertex kind vk_{min} must match, and there are few of them, it makes sense to check subgraph isomorphism only near them. It is possible for the neighbor graphs to overlap, in which case some PDG nodes appear in multiple distinct neighbor graphs and will be tested for isomorphism with the Bug PDG multiple times.

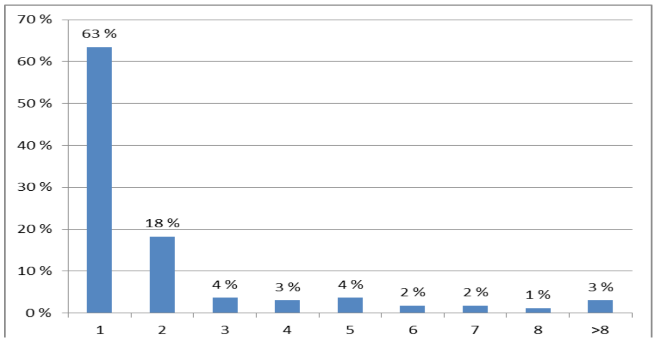

Figure 2. Size (contiguous lines) of the largest component of each bug fix

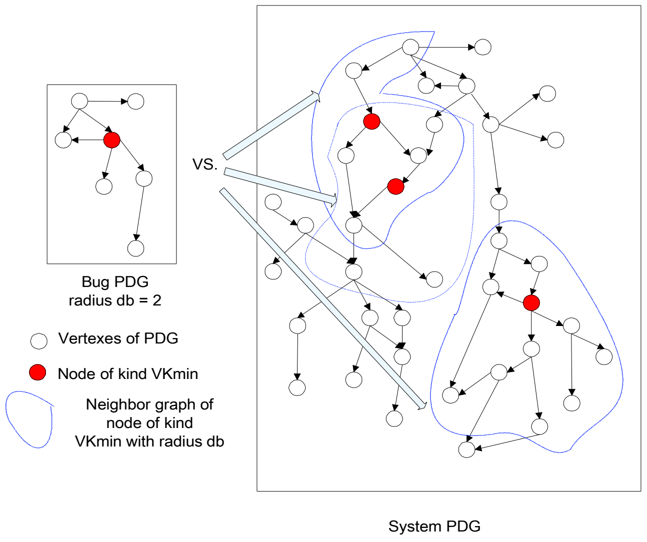

Figure 3. Breaking the System PDG into smaller pieces (Opt2)

Opt2 adds some extra overhead to CBCD. Here is the theoretical analysis of the time complexity without Opt2 and with Opt2. We assume that the Bug PDG has i_1 nodes and j_1 edges and the System PDG has i_2 nodes and j_2 edges. Then the time complexity of each step of Opt2 is:

- Opt2-step1. $O(i_1+i_2)$
- Opt2-step2. $O(1)$
- Opt2-step3. $O(i_1\ j_1)$, because of the igraph_diameter() function of igraph [16].
- Opt2-step4: $O(w(i_2+j_2))$, where there are w vertexes in the System PDG having the chosen vertex kind from Opt2-step2, because of igraph_neighborhood_graph() function of igraph [16] .

Although Opt2 adds the above overhead, it can significantly reduce the time complexity of Step 3 of Section III.B, i.e. subgraph isomorphism matching.

Without Opt2, the time complexity of comparing the Bug PDG and the System PDG is between $O((i_1+ j_1+ i_2+ j_2)^2)$ and $O((i_1+ j_1+ i_2+ j_2)!\ (i_1+ j_1+ i_2+ j_2))$, for the algorithm [12] implemented by igraph.

Since each subgraph of the System PDG has identical pseudo-radius as the Bug PDG after Opt2, we can assume the size of subgraph of the System PDG is $v(i_1+j_1)$, where v is expected to be close to 1. With Opt2, we compare the Bug PDG with w neighbor graphs in the System PDG in Step 3 of CBCD. The time complexity of each comparison will be between $O(w(i_1+j_1+v(i_1+j_1))^2)$ and $O(w(i_1+j_1+v(i_1+j_1))!\ (i_1+j_1+v(i_1+j_1)))$.

Let us compare the time complexity of isomorphism testing without Opt2 with Opt2:

- The best case:
  $O(w(i_1+j_1+v(i_1+j_1))^2)$ vs. $O((i_1+ j_1+ i_2+ j_2)^2)$
- The worst case:
  $O(w(i_1+j_1+v(i_1+j_1))!\ (i_1+j_1+v(i_1+j_1)))$ vs.
  $O((i_1+ j_1+ i_2+ j_2)!\ (i_1+ j_1+ i_2+ j_2))$

Opt2-step2 chooses the vertex kind with the fewest occurrences. So, it reasonable to assume that w is small, namely much less than i_2. In addition, we have observed that the buggy code often includes only a few lines, so we can assume i_1+j_1 is much smaller than i_2+j_2. If the two assumptions stand, the time complexity of comparing the Bug PDG and System PDG with Opt2 will be at least as good as the time complexity of this step without Opt2 in the best case. Even in the worst case, the time complexity with Opt2 will still be better than the one without it, because i_1+j_1 is related to the size of the buggy code, which is often small, while i_2+j_2 is related to the size of the procedure to be compared, which can have hundreds of lines of code.

3) *Optimization 3 (Opt3): Exclude Irrelevant PDGs*

This optimization discards some parts of the System PDG. The Bug PDG must match within one of the (relatively small) components of the System PDG. More specifically, each node of the Bug PDG must correspond to some node of a System PDG component, so each System PDG component must have as many, or more, nodes of each vertex kind than the Bug PDG does. CBCD discards any System PDG component that does not satisfy this criterion.

For example, suppose the Bug PDG has four nodes of the "expression" vertex kind, two nodes of the "control-point" vertex kind, and two nodes of the "actual-in" vertex kind. If a System PDG component includes four nodes of the "expression" vertex kind, one node of the "control-point" vertex kind, and three nodes of the "actual-in" vertex kind, this System PDG component will be excluded from isomorphism matching, because it has too few nodes of vertex kind "control-point". It therefore cannot be a supergraph of the Bug PDG.

4) *Optimization 4 (Opt4): Break Up Large Bug Code Segments*

Although most bug segments cover 4 or fewer lines of contiguous code, as shown in Fig. 2, some bug segments are larger. When the buggy code segment is large, Opt1, Opt2, and Opt3 may not be able to improve the performance of the system enough, because:

- When the buggy code segment is large, the Bug PDG will include many vertex kinds. Thus, Opt1 may not be able to prune many edges of the System PDG.
- When the buggy code segment is large, the radius of the Bug PDG will be large. Thus, the sub-graphs of the System PDG after Opt2 will still be large and isomorphism matching will be slow.
- Even if few large Bug PDGs and large System PDGs need to be compared for isomorphism matching, the system will perform very slowly. Thus, Opt3, which reduces the number of comparisons, does not help enough.

To deal with large contiguous buggy code, we implemented a fourth optimization. It is only triggered when the bug has more than 8 lines of contiguous code. The optimization is performed in Step 2 of CBCD and breaks up bug code segments into sub-segments with fewer lines of code. We set two thresholds, which are configurable and default to 4 and 6. The purpose of setting these two thresholds is to split large buggy code segment into smaller sub-segments, and at the same time avoid having too small sub-segments. For a buggy code segment having more than 8 lines of code, CBCD puts the first 4 lines of code in a sub-segment first. If the remaining lines have 6 or few lines of code, CBCD does not split it further. Otherwise, CBCD again puts the first 4 lines of the remaining lines in the second sub-segment and reconsiders the remaining lines. CBCD searches for clones of each sub-segment independently, and then merges their corresponding matched clones together. Merging can increase the false positive rate of CBCD, if CBCD merges two unrelated partial matches into a "complete" match that it would never have discovered if using the larger bug PDG. To deal with this issue, CBCD checks the last line of one suspected buggy sub-segment with the first line of another suspected buggy sub-segment to be merged. If the difference is more than 8 lines of code or the two sub-segments are in different files, CBCD assumes that these two code lines are too far apart to be part of clone of a single bug and does not merge them.

IV. Evaluation And Discussion

We wished to answer the following research questions:

- How well can CBCD find cloned buggy code?
- How well does CBCD scale?

A. *The Subject Programs*

We evaluated CBCD on Git, the Linux kernel, and PostgreSQL. We chose those three systems because:

- They are programmed mainly using C/C++, which means that they can be compiled by CodeSurfer.
- Their revision histories enable us to find buggy code and cloned buggy code for our evaluation.
- Git has more than 100K lines of code, PostgreSQL has more than 300K lines of code, and the Linux kernel has millions of lines of code, making them a good test of the scalability of CBCD.

B. *Evaluation Procedure*

1) *Oracles for the Evaluation*

As discussed in Section III.B, determining true clones of buggy code is undecidable. Our experiments use as an oracle the clones of buggy code that developers identified. It is possible that the developers found only some clones of a given bug, in which case any tool that reported the others would be (incorrectly) considered to suffer false positives.

As described in Section II, we identified buggy code and its clones by searching commit logs and reading code. From these bugs, we chose only those related to C/C++ code, because that is the only type of code that CodeSurfer can compile. We examined all 12 Git bugs and all 22 PostgreSQL bugs from Table III, and we arbitrarily chose 52 (one third of 157) Linux bugs from Table I. We were not able to use all of these bugs: our technique is not applicable when the bug fix adds new code; CBCD only handles C and C++; our processor is 32-bit x86; and in two cases the developers were mistaken in calling two bugs clones, because they refer to completely different functions or data structures (see Table V). After excluding such cases, the evaluation used 5 Git bugs, 14 PostgreSQL bugs, and 34 Linux bugs. A complete list of the bug clones examined in the evaluation is in [24].

TABLE V. BUGGY CODE THAT PROGRAMMERS CALLED "CLONES" BUT ARE NOT TRUE CLONES

Buggy lines of code	**Not identical code under CBCD definition**
struct **lock_file** packlock;	struct **cache_file** cache_file;
if (**ahd_match_scb**(ahd, pending_scb, scmd_id(cmd))	if (**ahc_match_scb**(ahc, pending_scb, scmd_id(cmd))

2) Other Code Clone Detectors for Comparison

To compare CBCD with other types of code clone detectors, we also ran Simian v2.3.32 [25] (text-based), CCFinder v10.2.7.3 [1] (token-based), Deckard v1.2.1 [6] (AST-based), and CloneDR v2.2.5 [26] (AST-based) on these 53 bugs.

These code clone detectors favor large cloned code segments rather than small ones. As shown in Fig. 2, cloned bugs are mostly less than 4 lines of code, so we adjusted some parameters to make the code clone detectors work better. For Simian, we set the number of lines of code to be compared for clones to its minimum value, i.e. 2, and used default values for the other parameters. For CCFinder, we set the minimum clone length to be 10 and the minimum TKS to be 1. For Deckard, we set min_tokens to 3, stride to 2, and similiartiy threshold to 0.95. For CloneDR, we set the minimum clone mass to 1, the number of characters per node to 10, number of clone parameters to 5, and similarity threshold to 0.9.

For Simian, CCFinder, and Deckard, the system to be checked for buggy clones is the same file set as CBCD. However, CloneDR failed with parse errors when we input the same file set as for CBCD. To enable a comparison with CBCD, we used a "*slim evaluation"*: the "system" input to CloneDR is *only* the files that include the bug and the buggy clones found by CBCD. We additionally commented out lines that CloneDR could not parse. The slim evaluation determines whether CloneDR can find the clones that are identified by CBCD. However, the slim version includes only 2% of the input files and 1% of the lines of code. If CloneDR could run on all files, its false positive rate would be much higher than reported in the slim evaluation.

3) Executing the Tools

The input to each tool is: the file that contains the buggy code (along with the starting and ending lines of the buggy code segment, if the tool accepts it; only CBCD did), plus the system to be checked for buggy clones.

We recorded the execution time of CBCD using the Linux command "time". The evaluation was run on a PC with 4G memory, 3Ghz CPU, and running Ubuntu 10.04.

4) Metrics

A false negative is a clone identified by the developer but not identified by the tool. A false positive is a clone reported by a tool that the developers did not report as buggy.

We count a clone as found if a tool reports a clone pair whose parts are as large as, or larger than, the original buggy code and the developer-identified buggy clone. This metric is very generous to the other code clone tools. CBCD reports clones that have similar size to the buggy code. The other code clone tools report much larger clones, because they are designed for a different purpose: to find large cloned code segments. Often a single result subsumed several of CBCD's results. Such large results would be less useful to a programmer. These issues make a direct comparison of precision and recall, or of the exact number of true and false positives and negatives, misleading. Instead, for each tool, we categorized each of the 53 bugs as follows.

- **N1**: *no false positives, no false negatives.*
- **N2**: *no false positives, some false negatives*.
- **N3**: *some false positives, no false negatives.*
- **N4**: *some false positives, some false negatives.*

C. How Well Can CBCD Find Cloned Buggy Code?

Table VI counts the bugs in each category. CBCD outperforms the other tools in finding buggy clones correctly, i.e., CBCD has the highest number in N1. Deckard performs the worst, partially because it failed with parse errors in 15 out of the 29 N2 cases. Unlike CloneDR, Deckard does not report precisely the location of the parse error. Thus, we could not perform a slim evaluation as with CloneDR.

TABLE VI. COMPARISON WITH OTHER CODE CLONE DETECTORS

	CBCD	**Simian**	**CCFinder**	**Deckard**	**CloneDR -slim**
N1	36 (68%)	16 (30%)	24 (45%)	14 (26%)	31 (58%)
N2	6 (11%)	36 (68%)	11 (21%)	29 (55%)	14 (26%)
N3	11 (21%)	1 (2%)	12 (23%)	6 (11%)	7 (13%)
N4	0 (0%)	0 (0%)	6 (11%)	4 (8%)	1 (2%)

Researchers categorize code clones into four main types, and so-called "scenarios" subcategorize each type [27]. The distributions of our examined bugs are shown in details in [24] and are summarized as follows:

- 51% of duplicated bugs are Type-1: identical code fragments except for variations in whitespace, layout, and comments.
- 24% are in scenarios *a*, *b*, and *c* of Type-2: renaming identifiers or renaming data types and literal values. Most of the variable renaming is renaming of function actual arguments.
- 23% are in scenarios *a* and *b* of Type-3: small deletions or insertions.
- 2% are in scenario *a* of Type-4: reordering of statements.

The 5 tools perform about equally well on Type-1 and Type-2 clones. In theory, AST-based tools could be best on

Type-2 clones, but CBCD's text comparisons reduce its false positive rate in practice. CBCD outperforms all the other tools on Type-3 clones; for example, CBCD identifies the code segments shown in Table VII as clones while Simian, CCFinder, Deckard, and CloneDR suffer false negatives.

Unlike text-based, token-based, and AST-based clone detectors, a semantics-based clone detector like CBCD tolerates control-statement replacement. Our 53 examples did not include control-statement replacement (programmers might be less likely to call such code snippets "clones" in the bug tracking system), so we evaluated this claim by artificially modifying the code of a Git clone from a "for" statement to a "while" statement. The modified code is shown in Table VIII. CBCD identified the clone, but Simian, CCFinder, Deckard, and CloneDR did not.

TABLE VII. EXAMPLES OF BUGGY CLONES IDENTIFIED CORRECTLY BY CBCD BUT NOT BY OTHER CODE CLONE DETECTORS

Buggy lines of code	Bug clones
doorbell[0] = cpu_to_be32**((qp->rq.next_ind << qp->rq.wqe_shift) | size0)**;	doorbell[0] = cpu_to_be32**(first_ind << srq->wqe_shift)**;
ret = btrfs_drop_extents(trans, root, inode, start, **aligned_end**, start, &hint_byte);	ret = btrfs_drop_extents(trans, root, inode, file_pos, **file_pos + num_bytes,** file_pos, &hint);

TABLE VIII. ORIGINAL CODE VS. CODE AFTER CONTROL REPLACEMENT

Original code	Code after control replacement
for (j = first; j <= last; j++){ struct object_entry *child = objects + deltas[j].obj_no; if (child->real_type == OBJ_REF_DELTA) resolve_delta(child, &base_obj, obj->type); }	**j = first;** **while (j <= last){** struct object_entry *child = objects + deltas[j].obj_no if (child->real_type == OBJ_REF_DELTA) resolve_delta(child, &base_obj, obj->type); **j++; }**

The 6 clones out of 53 that are not identified by CBCD, i.e. the false negative cases, are in Table IX. CBCD misses the first three clones because CodeSurfer's PDG does not represent data structures and macros; this is not a reflection on our technique, but on our toolset. CBCD misses the last three clones because they include variable renaming in an expression. When a vertex in the PDG is recognized as "expression", as explained in Section III.C.1, CBCD compares the characters of the expression to avoid false positives.

All 11 bugs for which CBCD reports a false positive are similar: the buggy code is one line of code calling a function, or a few one-line function calls without data/control dependencies among them. For all 11 bugs, Simian, CCFinder, or Deckard either also report a false positive, or else suffer a false negative due to a built-in threshold that prevents them from ever finding any small clone. CloneDR-slim does slightly better, with 2 false negative and 7 false positives. Recall that we used a slim evaluation for CloneDR; if it ran on all files, its false positive rate would be higher.

One example of CBCD's 11 false positives is shown in Table X. Other calls of the same function, such as memset(ib_ah_attr, 0, sizeof param), are returned by CBCD, because it tolerates renaming of actual input and output parameters. However, as mentioned in Section IV.C.3, we count as a false positive any CBCD output that is not yet reported by the developers as buggy. Some of the CBCD-identified clones of the bug code segments might be bugs that have been overlooked by developers. Thus, CBCD's real false positive rate may be lower than Table VI reports.

TABLE IX. FALSE NEGATIVES: BUGGY CODE CLONES THAT ARE NOT IDENTIFIED BY CBCD

The bug fix shown by "diff"
static const struct amd_flash_info jedec_table[] = { - .devtypes = CFI_DEVICETYPE_X16| CFI_DEVICETYPE_X8, - .uaddr = MTD_UADDR_0x0555_0x02AA,
static struct ethtool_ops bnx2x_ethtool_ops = { - .get_link = ethtool_op_get_link,
#define desc_empty(desc) \ - (!((desc)->a + (desc)->b))
- obj = ((struct tag *)obj)->tagged; VS. - object = tag->tagged;
- blue_gain = core->global_gain + core->global_gain * core->blue_bal / (1 << 9); VS. - red_gain = core->global_gain + core->global_gain * core->blue_bal / (1 << 9);
- if (!hpet && !ref1 && !ref2) VS. - if (!hpet && !ref_start && !ref_stop)

TABLE X. EXAMPLES OF FALSE POSITIVES

Buggy code	All identified clones
memset(ib_ah_attr, 0, sizeof *path);	**True positive:** memset(ib_ah_attr, 0, sizeof *path); **False positive:** memset(best_table, 0, sizeof(best_table)); memset(best_table_len, 0, sizeof(best_table_len)); memset(p, 0, padding); etc.

Table XI shows another kind of code that might lead to potential false positive reports from CBCD. Fig. 4 shows the PDGs. The two vertexes representing "close()" in Bug PDG and the four vertexes representing "close()" in System PDG lead to several sub-graph isomorphism relationships between these two PDGs. Thus, CBCD returned several semantically identical correspondences between the buggy code and suspected code. However, all CBCD results point to the same suspected code. CBCD coalesces duplicate results that point to the same code location.

D. How Well Does CBCD Scale to Larger Bugs?

In our experiments, CBCD finished in seconds after CodeSurfer completed. However, this is not a good test of scalability, because the cloned bugs are often platform- or architecture-dependent, in which case the command line (in the developer-supplied Makefile) that compiles them does not compile the whole system.

To determine how well CBCD works with larger bug segments, we searched the Linux and Git SCM using the key word "duplicate". We chose four of these (non-buggy) code segments from Git and four from Linux. The four Linux code segments are located in subcomponents "net", "fs", "drivers", and "drivers" of Linux of different versions respectively, and we compiled the relevant subcomponent. For Git, we compiled the whole relevant version (Git changed size over time). Table XII gives the results.

TABLE XI. BUGGY CODE AND SUSPECTED CODE OF A POTENTIAL FALSE POSITIVE IN GIT

Buggy code	System code

if(pid! = 0){
close(fd[1]);
dup2(fd[0], 0);
close(fd[0]);}
	if(pid! = 0){
	close(fd[1]);
	dup2(fd[0], 0);
	close(fd[0]); }
	close(fd[0]);
	close(fd[1]);

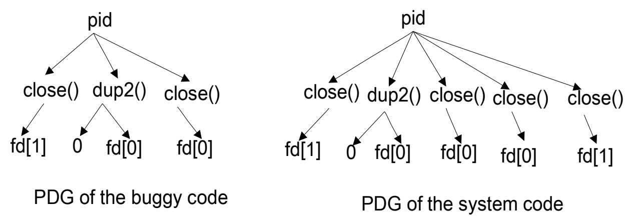

Figure 4. Snippet of the PDG of the buggy and system code in Table XI

Step 1 of CBCD (performed by CodeSurfer, version 2.1) takes a long time if the system is big, but this is done only once and can be reused. We expect CodeSurfer's performance to improve in later versions. Checking for clones of new bugs requires only running Step 2 and 3, which takes only seconds.

The running time of Simian, CCFinder, and Deckard using the same parameter setting as explained in Section IV.B are shown in Table XIII. We could not run CloneDR because of its parse errors.

CBCD is slower than Simian and Deckard if CBCD's preprocessing (Step 1) is included. Considering only the incremental cost of Steps 2 and 3, CBCD is competitive. Setting parameters to let CCFinder detect small clones makes it slower than CBCD, because generating all small clone pairs first, and then searching for clones of a certain code segment, is inherently inefficient. This could be changed, but CBCD is more accurate than the other approaches, regardless of their settings. We believe the cost of undetected bugs makes CBCD worth running even if all steps are required.

E. *Performance Improvement Due to the Four Optimizations*

We used four optimizations to speed up CBCD. We have examined the unique benefits of a given optimization that are not obtained by other optimizations. For example, to evaluate Opt2, we compared CBCD with Opts 1+3+4 against CBCD with Opts 1+2+3+4.

The results show that our optimizations can greatly improve the performance of the isomorphism matching by reducing the complexity and number of graphs to be compared. Detailed data are shown in Appendix C of [24].

Opt1, i.e. filtering out the irrelevant edges and vertexes in the System PDG, contributes most to the CBCD performance improvement. Opt1 pruned on average 90% of the edges before the subgraph isomorphism comparison. For the 53 bugs, Opt1 on average improved performance 622 times. However, the variation is high. One case achieved 20237 times performance improvement and another achieved 11890 times performance gain. In one of the four "duplicate code" Linux cases, without Opt1, the execution of the Step 3 of CBCD was aborted (igraph's [16] subgraph isomorphism function reported an out-of-memory error, because the System PDG is too big and too many isomorphic subgraphs are returned).

TABLE XII. RUNNING TIME OF EACH STEP OF CBCD

Id	NLOC / Number of PDG edge		CBCD steps		
	Sys.	*Bug*	*1*	*2*	*3*
Git-1	67K/358K	10/38	6m	**13s**	**5s**
Git-2	75K/441K	4/4	15m	**4s**	**2s**
Git-3	81K/414K	9/39	18m	**9s**	**3s**
Git-4	81K/414K	16/33	18m	**6s**	**2s**
Linux1	170K/1022K	6/70	32m	**15s**	**6s**
Linux2	140K/830K	3/3	25m	**16s**	**4s**
Linux3	363K/1970K	4/4	159m	**39s**	**8s**
Linux4	313K/1645K	3/13	95m	**17s**	**7s**

TABLE XIII. RUNNING TIME OF OTHER CLONE DETECTORS

Id	Simian	CCFinder	Deckard
Git-1	2s	5m	4m
Git-2	2s	6m	5m
Git-3	2s	8m	6m
Git-4	2s	8m	6m
Linux1	6s	63m	8m
Linux2	5s	34m	7m
Linux3	16s	899m	32m
Linux4	13s	623m	24m

Opt2, i.e. breaking the System PDG into smaller graphs, improves Step 3 of CBCD by 2 to 3 times. In one case, Opt2 improved performance by 72 times. The performance gain of Opt2 is not significant in other cases, because Opt1 prunes out most edges of the System PDG. In 90% of our examined cases, the average ratio of size (number of edges and vertexes) of subgraph of the System PDG to size of the Bug PDG, i.e. the "*v*" in the formulas of Section III.C.2, is less than 1.

Opt3, i.e. excluding irrelevant System PDGs, also improves Step 3 of CBCD by 2 to 3 times. As with Opt2, after Opt1 filters out most of the edges of the System PDG, few subgraphs of the System PDGs are left for comparison.

Opt4, i.e. breaking the large bug code segment, is applicable only to three clones that have more than 8 lines of code. In one case, Step 3 of CBCD sped up by 120 times, but the other two showed no significant performance improvement. Examination of these code segments shows that Opt4 can bring significant performance gains when the bug code segment has many vertex kinds, especially vertex kinds such as "actual_in", "actual_out", or "declaration", that are related to procedure parameters or arguments. In such cases, Opt1 cannot filter out many vertexes and edges of the System PDG. On the contrary, if the number of different vertex kinds of the Bug PDG is small, many vertexes and edges of other vertex kinds in System PDGs will be pruned out using Opt1, and Opt2 and Opt3 are also more effective, subsuming the benefits of Opt4.

F. Threats to Validity

1) Threats to Internal Validity

The buggy code used for evaluation consists of real cloned bugs in Git, the Linux kernel, and PostgreSQL, but were not chosen to be representative or comprehensive. We do not know how many cloned bugs these projects really have, but we do know that around 4% of the bugs in a commercial product were duplicates.

2) Threats to External Validity

We tested CBCD only on Git, the Linux kernel, and PostgreSQL. It is possible that other subject programs would have different characteristics. Furthermore, the evaluation considers only 53 cloned bugs in detail, and these were not chosen to be representative.

3) Threats to Construct Validity

To measure the false positive rate of CBCD, we used the clones identified by the developers as an oracle. As mentioned in Section IV.C, the developers might have overlooked some clones, so CBCD's real false positive rate may be lower than reported in this paper.

G. Application Constraints

Although bugs consisting of a one-line function cause false positives in our experiment, and Fig. 2 shows that most code fixes are on one line, this does not limit the applicability of CBCD. In real life, developers can often merge the buggy code line with few lines before or after it, which can be regarded as the context of the buggy code, to make a bigger code segment as the input for CBCD. This may help avoid false positives. We did not perform this in our experiments to avoid evaluation bias.

V. Related work

Previous code detection methods can be classified into:

- Token-based code clone detecting methods [1, 2] examine token sequence similarities.
- Text [3] or string-based [4] code clone detection methods compare the text or strings in the code.
- Abstract syntax tree (AST) based code clone detection methods [5, 6] match two ASTs to find code clones.
- PDG-based code clone detection tools [7, 8, 9, 10] try to overcome the limitations of the above code clone detectors by comparing the data and control dependence graphs of the code segments.
- Behavior-based code clone detection [32] tries to find code clone based on the execution results of test cases.
- Memory-state-based code clone detection [33] compares the abstract memory states of code.

Most previous code clone detection tools search for large clones for code refactoring or to find plagiarism. Thus, most such tools do not compare small code segments that span only a few lines. For example, PDGs smaller than a certain size are excluded from comparison in [9]. In general, such tools have no knowledge of which segment of code should be the input for clone searching. Thus, some of these tools start with the first line of the system, and extract 10 or 20 lines as input for searching for code clones.

We have identified a new, important use case. CBCD solves a different problem than scanning an entire codebase for plagiarism detection or identifying refactoring opportunities. CBCD is more like an advanced "find" command. The input is a small code segment that includes a few contiguous lines of code (most buggy segments cover only a few contiguous lines of code, unless the bug is caused by missing functionality or a design change). The outputs are all locations of the clones of such a code segment. A user might assume that general code clone detectors would also perform well at detecting clones of buggy code. However, as our evaluation showed, this assumption would be wrong. CBCD outperforms text-based, token-based, and AST-based clone detectors to find cloned buggy code, especially Type-3 and Type-4 clones. We did not compare CBCD with behavior-based clone detectors, because we lack detailed knowledge of the expected dynamic behavior of the buggy code. Memory-state-based clone detectors do not fit the purpose of detecting cloned buggy code.

Unlike generic code clone detectors; CBCD does not generate all code clone pairs in advance. It only searches for clones of a small code segment on demand. The rationale is that people are usually not interested in finding code clones of small code segments to refactor them. However, when they find that a code segment is buggy, they need to find all its clones and fix all of them. As mentioned in Section IV.B and IV.E, searching for clones on demand rather than generating all clone pairs at once makes CBCD more scalable than general clone detectors. But, even if other clone detectors adopted CBCD's incremental approach, CBCD is still more accurate.

CBCD uses PDG-based code clone detection principles to detect clones. PDG-based methods usually face scalability problems in sub-graph isomorphism checking. One proposed solution to improve the performance of PDG-based code clone detection is to match the PDG back to the AST [10], so that the graph isomorphism problem is simplified into a tree similarity problem. However, such a simplification excludes information for some edges in the PDG and makes the PDG comparison incomplete. Another proposed solution to the scalability problem is to compare the vertex histogram of PDGs first to exclude highly dissimilar PDGs and stop the sub-graph isomorphism matching after the first isomorphism is found [9]. Such a solution is lossy, because a dissimilar vertex histogram between a small PDG and a big PDG does not guarantee that the small PDG will not have a subgraph isomorphism relationship with the large PDG. A PDG-based code clone detector [7] based only on graph isomorphism performed poorly compared to other code clone detectors [30]. CBCD improves the accuracy of PDG-based code clone detection by utilizing the syntax and text information of the buggy code to prune and break the PDG to be compared. Compared to the system in [9], CBCD is less lossy and is more scalable to large PDGs. Yet another proposed solution to the scalability problem is to compare the PDG only within radius 5 of a vertex of "control-point" kind [19]. This is lossy and depends on hard-coded choices of radius and vertex kind; by contrast, our Opt2 is not lossy and is general.

The studies [28, 29] transform the code query into graph reachability patterns and match the patterns in the SDG of the source code. Such a method can potentially be used to detect clones of buggy code. However, developers must manually describe the buggy code using code query language. Compared to these methods, CBCD is easier to use, because it automatically transforms the buggy code into PDG graphs and then matches the buggy PDG with the PDG of the suspected code. Similarity, graph-matching algorithm has been used to match design patterns [34]. However, the algorithm in [34] is not directly applicable since it finds a hard-coded set of design patterns rather than clones of arbitrary bugs. CP-Miner [2] is a code clone detection tool that searches for bugs caused by code copy-paste. CP-Miner can only find "bugs caused when programmers forget to modify identifiers consistently after copy-pasting". The study [31] also compares tokens to search defect clones.

The SecureSync tool [18] is similar to CBCD, i.e. a tool to find duplications of a software vulnerability/bug. To use SecureSync, the clones must be classified into categories I, II, and III first. A category I code clone is due to code copy/paste. For such a code clone, an AST-based method is proposed. A category II code clone is due to function reuse. To detect such a clone, the local PDG around a function call is built and compared. All other code clones are categorized into III without any methods proposed to detect them. Compared to SecureSync, CBCD is easier to use. People do not need to categorize code clone into different categories and treat them differently. For category I code clones, CBCD better tolerates code insertion, deletion, and re-ordering. CBCD can potentially support more kinds of code clone, for example, those in category III of SecureSync. We would like to compare CBCD with SecureSync [18], but according to its authors, SecureSync is not available for public distribution yet. Jiang et al. [20] investigated how to discover clone-related bugs through comparing the nodes in parse trees. In [21], the attributes of edges and nodes of two graphs are extracted to optimize the performance of graph isomorphism comparison for detecting clones of MATLAB/Simulink models. In [22], 17-45% of bug-fixing changes were found to be recurring, and most of them occurred in multiple files at the same revision (i.e. in space). However, this study targets identifying bug clones in object-oriented systems. In [23], a few clone detection algorithms are combined with parallel algorithm to detect buggy inconsistency in a very large system.

VI. Conclusions and Future Work

We have identified a new, important use case for code clone detection (finding buggy clones), motivated its importance in real-world systems, given an algorithm for finding buggy clones, and evaluated its accuracy and performance. Whereas previous work was motivated by code refactoring or plagiarism detection, we focus on detecting cloned buggy code.

The contributions of our work include:

1. We examined real-world bug reports and SCM data, and established that identical (cloned) bugs are a serious problem. In a commercial product line, cloned bugs were common and important, comprising 4% of all bugs.

2. We proposed a methodology for improving system reliability: After a bug is fixed, the programmer should search for other code that behaves similarly to the detected buggy lines. Even if a system has relatively few cloned bugs, finding these bugs is valuable for programmers and can be done relatively accurately and inexpensively.

3. We extended previous PDG-based clone detection algorithms to make them more scalable, by pruning the search space of sub-graph isomorphism matching. Detecting small clones required different algorithms and implementations than previous code detectors, which are less effective in finding bug clones.

4. We implemented our algorithms in a tool, CBCD, that detects possible clones of buggy code by comparing the Bug PDG and the System PDG. The CBCD tool is available on request for research purposes.

5. We evaluated CBCD with known cloned bugs and known cloned lines of code, showing that CBCD is scalable and effective in searching for possible clones of buggy code. Other clone detection tools are less effective for this purpose.

The performance bottleneck of CBCD is CodeSurfer's PDG generation. Future work is to improve performance of this step to make CBCD even more scalable.

Acknowledgments

This work was supported in part by grant #183235/S10 from the Norwegian Research Council, by the JIP partners, and by US NSF grant CCF-1016701.

References

[1] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code," IEEE Trans on Software Engineering, vol. 28, no. 7, pp. 654-670, July 2002.

[2] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code," IEEE Trans on Software Engineering, vol. 32, no. 3, pp. 176-192, March 2006.

[3] S. Ducasse, M. Rieger, and S. Demeyer, "A Language Independent Approach for Detecting Duplicated Code," Proc. IEEE intl. conf. on Software Maintenance (ICSM'99), IEEE Press, Sept. 1999, pp. 109-118.

[4] B. S. Baker, "On Finding Duplication and Near-duplication in Large Software Systems," Proc. the Second Working Conference on Reverse Engineering, IEEE Press, July 1995, pp. 86-95.

[5] R. Koschke, R. Falke, and P. Frenzel, "Clone Detection Using Abstract Syntax Suffix Trees," Proc. the 13th Working Conference on Reverse Engineering, IEEE Press, Oct. 2006, pp. 253-262.

[6] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones," Proc. Intl. conf. on Software Engineering (ICSE'07), IEEE Press, May 2007, pp. 96-105.

[7] J. Krinke, "Identifying Similar Code with Program Dependence Graphs," Proc. the 8th Working Conference on Reverse Engineering (WCRE'01), IEEE Press, Oct. 2001, pp. 301-309.

[8] R. Komondoor and S. Horwitz, "Using Slicing to Identify Duplication in Source Code," Proc. the 8th International Symposium on Static Analysis (SAS' 01), Spring-Verlag Press, July 2001, pp. 40-56.

[9] C. Liu, C. Chen, J. Han, and P. S. Yu, "GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis," Proc. 12th ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining, ACM Press, Aug. 2006, pp. 872-881.

[10] M. Gabel, L. Jiang, and Z. Su, "Scalable Detection of Semantic Clones," Proc. Int. Conf. on Software Engineering (ICSE'08), ACM Press, May 2008, pp. 321-330.

[11] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, “An Empirical Study of Operating Systems Errors,” Proc. the 8th ACM Symp. on Operating Systems Principles, ACM Press, Oct. 2001, pp. 73-88.

[12] L. P. Cordella, P. Foggia, C. Sansone, and M. A. Vento, “(Sub)Graph Isomorphism Algorithm for Matching Large Graphs,” IEEE Trans on Pattern Analysis and Machine Intelligence, vol. 26, no. 10, pp. 1367-1372, Oct. 2004.

[13] R. C. Read, and D. G. Corneil, “The Graph Isomorphism Disease,” Journal of Graph Theory, vol. 1, no. 4, pp. 339–363, Winter 1977.

[14] CodeSurfer: http://www.grammatech.com/products/codesurfer/overview.html

[15] J. Ferrante, K. J. Ottenstein, and J. D. Warren, “The Program Dependence Graph and its Use in Optimization,” ACM Trans on Programming Languages and Systems, vol. 9, no. 3, pp. 319-349, July, 1987.

[16] G. Csárdi and T. Nepusz, “The Igraph Software Package for Complex Network Research,” InterJournal Complex Systems, 2006, pp. 1695.

[17] B. D. McKay, “Practical Graph Isomorphism,” Congressus Numerantium, 30 (1981), pp. 45-87.

[18] N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, “Detection of Recurring Software Vulnerabilities,” Proc. Intl. Conf. on Automated Software Engineering (ASE’10), ACM Press, Sept. 2010, pp. 447-456.

[19] R.-Y. Chang, A. Podgurski and J. Yang, “Discovering Neglected Conditions in Software by Mining Dependence Graphs,” IEEE Trans on Software Engineering, vol. 34, no. 5, pp. 579-596, Sept. 2008.

[20] L. Jiang, Z. Su, and E. Chiu, “Context-based Detection of Clone-related Bugs,” Proc. 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symp. on The foundations of software engineering (ESCE/FSE’07), ACM Press, Sept. 2007, pp. 55-64.

[21] N. H. Pham, H. A. Nguyen, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen, “Complete and Accurate Clone Detection in Graph-based Models,” Proc. Intl. Conf. on Software Engineering (ICSE’09), IEEE Press, May 2009, pp.276-286.

[22] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, “Recurring Bug Fixes in Object Oriented Programs,” Proc. Intl. Conf. on Software Engineering (ICSE’10), ACM Press, May 2010, pp. 315-324.

[23] M. Gabel, J. Yang, Y. Yu, M. Goldszmidt, and Z. Su, “Scalable and Systematic Detection of Buggy Inconsistencies in Source Code,” Proc. ACM intl. conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA’10), ACM Press, Oct. 2010, pp. 175-190.

[24] J. Li, and M. D. Ernst, “CBCD: Cloned Buggy Code Detector,” Technical Report UW-CSE-12-03-20, 2012.

[25] Simian- Similarity Analyser: http://www.harukizaemon.com/simian/

[26] CloneDR: http://www.semdesigns.com/Products/Clone/

[27] C. K. Roy, J. R. Cordy, and R. Koschke, “Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach,” Sci. Comput. Program, vol. 74, no. 7, pp. 470-495, May 2009.

[28] X. Wang, D. Lo, J. Cheng, L. Zhang, H. Mei, and J. X. Yu, “Matching Dependence-related Queries in the System Dependence Graph,” Proc. Intl. Conf. on Automated Software Engineering (ASE’10), ACM Press, Sept. 2010, pp. 457-466.

[29] M. Martin, B. Livshits, and M. S. Lam, “Finding Application Errors and Security Flaws using PQL: a Program Query Language,” Proc. ACM intl. conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA’05), ACM Press, Oct, 2005, pp. 365-383.

[30] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, E. Merlo, “Comparison and Evaluation of Clone Detection Tools,” IEEE Trans on Software Engineering, vol. 33, no. 9, pp. 577-591, Sept. 2007.

[31] S. Bazrafshan, R. Koschke, and N. Gode, “Approximate Code Search in Program Histories,” Proc. 18th Working Conference on Reverse Engineering, in in press, 2011.

[32] L. Jiang and Z. Su., “Automatic Mining of Functionally Equivalent Code Fragments via Random Testing,” Proc. 8th Intl. Symp. on Software Testing and Analysis (ISSTA '09), ACM Press, July 2009, pp. 81-92.

[33] H. Kim, Y. Jung, S. Kim, and K. Yi, "MeCC: Memory Comparison-Based Clone Detector," Proc. 33rd Intl. Conf. on Software engineering (ICSE '11), ACM press, May 2011, pp. 301-310.

[34] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, “Design Pattern Detection Using Similarity Scoring,” IEEE Trans. On Software Engineering, vol. 32, no. 11, pp. 896-909, Nov. 2006.