Context-Based Detection of Clone-Related Bugs[*]

Lingxiao Jiang Zhendong Su Edwin Chiu
Department of Computer Science
University of California, Davis
{lxjiang, su, eychiu}@ucdavis.edu

ABSTRACT

Studies show that programs contain much similar code, commonly known as *clones*. One of the main reasons for introducing clones is programmers' tendency to copy and paste code to quickly duplicate functionality. We commonly believe that clones can make programs difficult to maintain and introduce subtle bugs. Although much research has proposed techniques for detecting and removing clones to improve software maintainability, little has considered how to detect latent bugs introduced by clones. In this paper, we introduce a general notion of *context-based inconsistencies* among clones and develop an efficient algorithm to detect such inconsistencies for locating bugs. We have implemented our algorithm and evaluated it on large open source projects including the latest versions of the Linux kernel and Eclipse. We have discovered many previously unknown bugs and programming style issues in both projects (with 57 for the Linux kernel and 38 for Eclipse). We have also categorized the bugs and style issues and noticed that they exhibit diverse characteristics and are difficult to detect with any single existing bug detection technique. We believe that our approach complements well these existing techniques.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*code inspections and walk-throughs, debugging aids*

General Terms: Experimentation, Reliability

Keywords: context-based bug detection, code clone-related bugs, inconsistencies, code clone detection

1. INTRODUCTION

Software projects contain much similar code (*i.e.*, code clones), which may be introduced by many commonly adopted software development practices, such as reusing a generic framework, following a specific programming pattern, and directly copying and pasting code. These practices can improve the productivity of software development by quickly replicating similar functionalities. However, such practices, especially copying and pasting, can also reduce program maintainability and introduce subtle programming errors. For example, when enhancements or bug fixes are done on a piece of duplicated code, it is often necessary to make similar modifications to the other instances of the code. As previous work [24] indicates, it is easy for developers to miss some instances of the duplicated code and thus to introduce subtle bugs. "I think I have fixed the bug. Why is it still happening?" and "Why does the function work well in that way, but not in this way?" may be example questions that software maintainers ask and which may allude to clone-related bugs.

Finding similar code automatically is an important step to alleviate the aforementioned issues. Much work [7, 17, 18, 25] has been done on clone detection. Also, many techniques [16, 28] have been proposed for eliminating similar code to help reduce software maintenance cost. On the other hand, various studies [19, 20, 28] indicate that similarity in software is inherent, and clone unification and removal may not always be desired. The reasons include (1) *limited expressiveness of programming languages*: clone instances may have evolved over a substantial period of time with many independent changes so that they cannot be easily unified or removed; (2) *performance concerns*: unified code may have worse performance; and (3) *software development practices*: some experimental code may not be appropriate to be unified. Thus, code clones would always exist, and clone-related bugs may also lurk around in mature code. Therefore, we need automatic techniques to detect and eliminate such errors.

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

Figure 1 shows several inconsistencies detected by our approach among similar code. In the first pair (# 1) of code snippets, lines 408–419 and lines 323–334 are detected as similar code. However, the enclosing `if` statements of the two pieces of code are different: one uses `strncmp` which takes three arguments, while the other uses `strcmp` which takes only two arguments in their respective conditions for `if`. This turns out to be a logic error in "Code 2."

For the second pair (# 2) of code snippets, lines 4861–4864 and lines 2386–2391 are detected as similar code. Their main difference is that two more statements (lines 2389–2390) in "Code 2" are enclosed in the `for` statement. One can see that `msgbuf[0]` in "Code 2" is always `NULL` (*i.e.*, `'\0'`), and thus nothing in `msgbuf`

[*]This research was supported in part by NSF NeTS-NBD Grant No. 0520320, NSF CAREER Grant No. 0546844, NSF CyberTrust Grant No. 0627749, and a generous gift from Intel. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ESEC/FSE'07, September 3–7, 2007, Cavtat near Dubrovnik, Croatia.
Copyright 2007 ACM 978-1-59593-811-4/07/0009 ...$5.00.

#	Code 1	Code 2 (Similar to Code 1 but buggy)

1:	File: linux-2.6.19/drivers/scsi/arm/eesox.c
	
	`407: if (length >= 9 &&` **strncmp** `(buffer, "EESOXSCSI", 9) == 0) {`
	`408:    buffer += 9;`
	`409:    length -= 9;`
	`410:`
	`411:    if (length >= 5 && strncmp(buffer, "term=", 5) == 0) {`
	`        ......`
	`418:    } else`
	`419:        ret = -EINVAL;`
	`420: } else`
	`421:    ret = -EINVAL;`
		File: linux-2.6.19/drivers/scsi/arm/cumana_2.c
		
		`322: if (length >= 11 &&` **strcmp** `(buffer, "CUMANASCSI2") == 0) {`
		`323:    buffer += 11;`
		`324:    length -= 11;`
		`325:`
		`326:    if (length >= 5 && strncmp(buffer, "term=", 5) == 0) {`
		`        ......`
		`333:    } else`
		`334:        ret = -EINVAL;`
		`335: } else`
		`336:    ret = -EINVAL;`

2:	File: linux-2.6.19/drivers/cdrom/sbpcd.c
	
	`4859: if (cmd_type==READ_M2)`
	`4860: {`
	`4861:  for (xa_count=0;xa_count<CD_XA_HEAD;xa_count++)`
	`4862:    sprintf(&msgbuf[xa_count*3], "␣%02X", ...);`
	`4863:  msgbuf[xa_count*3]=0;`
	`4864:  msg(DBG_XA1,"xa␣head:%s\n", msgbuf);`
	`4865: }`
		File: linux-2.6.19/drivers/cdrom/sbpcd.c
		
		`2386: for (i=0;i<response_count;i++)`
		`2387:` **{**
		`2388:    sprintf(&msgbuf[i*3], "␣%02X", ...);`
		`2389:    msgbuf[i*3]=0;`
		`2390:    msg(DBG_SQ1,"cc_ReadSubQ:%s\n", msgbuf);`
		`2391:` **}**

3:	File: gcc-4.0.1/gcc/fortran/dependency.c
	
	`414: if (l_stride != NULL)`
	`415:    mpz_cdiv_q (X1, X1,` **l_stride** `->value.integer);`
		File: gcc-4.0.1/gcc/fortran/dependency.c
		
		`422: if (l_stride != NULL)`
		`423:    mpz_cdiv_q (X2, X2,` **r_stride** `->value.integer);`

Figure 1: Sample context-based inconsistencies among similar code.

would be output. Although the difference does not significantly impact the functionality of the code, it is still a bug and would manifest in debugging code.

As another example, the third pair (# 3) of code snippets have difference in variable naming, which is a local inconsistency within the clones themselves. In particular, the `if` condition performs a `NULL` check on `l_stride`, but `r_stride` is used within the `if` statement in "Code 2." This is suspicious and indeed, it has been confirmed by the GCC developers as a bug and fixed quickly.

Although such bugs may be discovered by thorough testing, designing "enough" test cases is often difficult and time consuming. In addition, even if a program exhibits abnormal behavior, it may still require much time to locate the actual bug locations. Such bugs may also be difficult to detect using standard program analysis techniques: (1) These techniques usually require certain property specifications (*e.g.*, null-pointers cannot be dereferenced, array accesses must be within bound, and certain temporal safety properties should hold), but clone-related bugs are diverse and difficult to specify (*cf.* Section 3); and (2) Most of these techniques still have limited scalability, especially for code bases with millions of lines of code, such as the Linux kernel and Eclipse.

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

Contributions. This paper makes the following contributions:

- It introduces a general notion of context-based inconsistencies among similar code and presents an efficient algorithm to detect such inconsistencies for locating clone-related bugs;
- It presents a series of classification and filtering heuristics to rank inconsistencies based on their potential relations with actual bugs;
- It presents a detailed empirical evaluation of the approach on large open source projects, including the Linux kernel and Eclipse. Our approach has revealed many previously unknown bugs and programming style issues; many reported bugs have been confirmed and fixed by the developers; and
- It presents a thorough categorization of the discovered bugs and programming style issues, confirming the hypothesis that code cloning may introduce subtle errors and revealing the diverse characteristics of the bugs. Such a categorization may help developers avoid similar errors in the future.

[1] We use a tree-based clone detection tool, Deckard [17], in our implementation. However, any other clone detection tool may be used for this purpose.

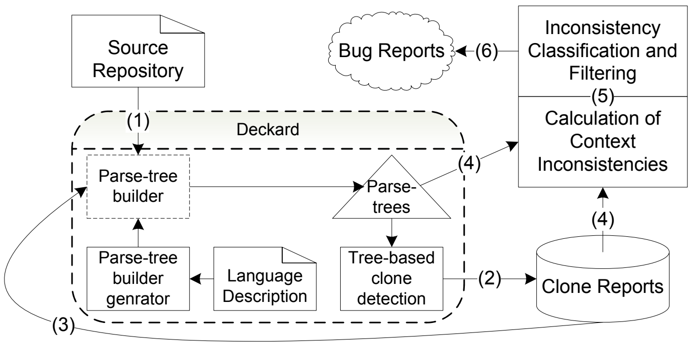
Source Repository
Bug Reports
(6)
Inconsistency Classification and Filtering
(5)
Calculation of Context Inconsistencies
(1)
Deckard
(4)
Parse-tree builder
Parse-trees
(4)
Parse-tree builder genrator
Language Description
Tree-based clone detection
(2)
Clone Reports
(3)

Figure 2: Overview of our bug detection approach.

2. ALGORITHM DESCRIPTION

In this section, we describe the details of our approach: (1) we first give a few basic definitions related to clones: *clone*, *clone pair*, *clone group*, and *contexts* (Section 2.1); (2) we then define three types of *context-based inconsistencies* among clones (Section 2.2); and (3) we next classify these inconsistencies based on their potential relations with actual bugs (Section 2.3) and present heuristics for pruning uninteresting ones (Section 2.4). The remaining clones with un-filtered inconsistencies may indicate bugs and are reported to developers for inspection.

2.1 Basic Definitions

For the definitions below, we assume there is a generic clone detection algorithm A such that $A(F_1, F_2) = true$ if and only if code fragments F_1 and F_2 are similar code w.r.t. a suitable definition of similarity (*e.g.*, in terms of tree editing distance [17]).

Definition 2.1 (Clones). A pair of code fragments F_1 and F_2 is called a *clone pair* if they are similar, *i.e.*, $A(F_1, F_2)$ holds. A group of code fragments $\{F_1, \ldots, F_k\}$ is called a *clone group* if $A(F_i, F_j)$ holds for all $1 \leq i, j \leq k$. Each code fragment F_i in a clone pair or a clone group is called a *clone instance*.

Definition 2.2 (Context). The *context* of a code fragment F is the innermost language construct that encloses F. We further restrict contexts to control-flow constructs. For example, in C, `if`, `switch`, `for`, `while` statements, and function definitions are such constructs. In Java, class definitions can also be such constructs.

We use the contexts for clones as the basis for inconsistency and bug detection to capture our intuition that similar code should perform similar functionalities and should be used under similar contexts. Thus clones with different contexts indicate likely bugs. Admittedly, the actual code surrounding clones may vary, and not all differences in the surrounding code are equally indicative of bugs. Thus, in this paper, we confine the context to be the smallest enclosing construct that may impact the control flows of a clone to ignore context differences that may be too far away from the clone.

We use an example to illustrate our definitions here: in the first pair of code snippets in Figure 1, the lines 408–419 and 323–334 are a clone pair, and the two `if` statements beginning with lines 407 and 322 are the respective contexts for the two clones.

Although the definition of a context is language-dependent, it is still straightforward to provide a generic algorithm to find the context of a given clone. Algorithm 1 gives the high-level description how we find the context of a clone. Given the parse tree of the program file which contains the clone, we perform a bottom-up search in the tree to find the smallest enclosing tree node of the clone that is a contextual node, *i.e.*, a control-flow construct.

2.2 Context-Based Inconsistencies

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

Definition 2.3 (Type-1 Inconsistency). Given a pair of clones F_1 and F_2 and their corresponding contexts C_1 and C_2, F_1 and F_2 have a *type-1 inconsistency* if the kinds of C_1 and C_2 (denoted by $\textsc{kind}(C_1)$ and $\textsc{kind}(C_2)$), in terms of language constructs, are different. We denote such an inconsistency by $I_1(F_1, F_2)$ such that $I_1(F_1, F_2) = 1$ if $\textsc{kind}(C_1) \neq \textsc{kind}(C_2)$, and $I_1(F_1, F_2) = 0$ otherwise.

We lift this definition to a clone group. Given a clone group G, there exists a unique equivalence partition $G/I_1 = \{g_1, g_2, \ldots, g_k\}$ of G such that (1) $\forall i \;\; (\forall C, C' \in g_i \;\; I_1(C, C') = 0)$, and (2) $\forall i \neq j \;\; (\forall C \in g_i \;\forall C'' \in g_j \;\; I_1(C, C'') = 1)$. We say that G has *type-1 inconsistency* if $k > 1$ and let $I_1(G) = k$ denote the type-1 inconsistency of G.

As an example, Figure 3 shows a clone pair which has type-1 inconsistency. The lines 3559–3567 and 2707–2715 are reported as a clone pair, and the context for "Code 1" is a function definition, while the context for "Code 2" is an `if` statement. In fact, the inconsistency was confirmed as a bug on line 3558: the developers omitted the necessary checks to make sure that `0` is a valid subscript and `item` is not `null`.

Calculating type-1 inconsistencies is as straightforward as finding contexts: we simply compare the kinds of the nodes returned by Algorithm 1. Despite their simplicity, type-1 inconsistencies have interesting potentials for finding many bugs, especially bugs due to missing checks (*cf.* Table 4).

Algorithm 1 Find the context of a given clone

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

Definition 2.4 (Type-2 Inconsistency). Given a pair of clones F_1 and F_2 and the conditional predicates P_1 and P_2 in their contexts, F_1 and F_2 have a *type-2 inconsistency* if P_1 does not match P_2 in terms of parse tree matching. We denote such an inconsistency by $I_2(F_1, F_2)$ such that $I_2(F_1, F_2) = 1$ if P_1 and P_2 do not match, and $I_2(F_1, F_2) = 0$ otherwise. If F_1 or F_2 has no corresponding predicates, we let $I_2(F_1, F_2) = 0$.

We lift this definition to a clone group. Given a clone group G, there exists a unique partition $G/I_2 = \{g_0, g_1, g_2, \ldots, g_k\}$ of G such that (1) g_0 contains exactly those clones with no context predicates, (2) $\forall i \neq 0 \;\; (\forall C, C' \in g_i \;\; I_2(C, C') = 0)$, and (3) $\forall i \neq j \neq 0 \;\; (\forall C \in g_i \;\forall C'' \in g_j \;\; I_2(C, C'') = 1)$. We say that G has *type-2 inconsistency* if $k > 1$ and let $I_2(G) = k$ denote the type-2 inconsistency of G.

This definition is also language-dependent because different languages may have different definitions of conditional predicates. As an example, the first pair of code snippets in Figure 1 has no type-1 inconsistency because both of them are `if` statements. However, they have a type-2 inconsistency because their `if` conditions invoke two different functions with different numbers of parameters.

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

A simple way to compare two conditional predicates P_1 and P_2 is to compare every node in the parse trees for P_1 and P_2 in a pre-order traversal. Although such a strict comparison may falsely report inconsistencies on semantically equivalent but syntactically different expressions, such as `p[i]` and `*(p+i)` in C, it may provide a reasonable upper-bound estimation on the total number of type-2 inconsistencies in clone groups. One can also argue that as long as the purpose of duplicating code is to improve software productivity (instead of plagiarism), code clones with the same semantics should often have the same syntactic structure and there is usually no reason to modify the code to have different syntactic structures. In practice, some code may become similar due to other reasons besides direct copying and pasting (*e.g.*, applying a same programming pattern). Thus semantically equivalent but syntactically different expressions do exist, and in Section 2.4, we employ certain heuristics to reduce false alarms on type-2 inconsistencies.

Definition 2.5 (Type-3 Inconsistency). Given a pair of clones F_1 and F_2, F_1 and F_2 have a *type-3 inconsistency* if F_1 and F_2 contain different numbers of *unique* identifiers. We denote such an inconsistency by $I_3(F_1, F_2)$ such that $I_3(F_1, F_2) = 1$ if F_1 and F_2 have different numbers of unique identifiers, and $I_3(F_1, F_2) = 0$ otherwise.

We lift this definition to a clone group. Given a clone group G, there exists a unique equivalence partition $G/I_3 = \{g_1, g_2, \ldots, g_k\}$ of G such that (1) $\forall i \;\; (\forall C, C' \in g_i \;\; I_3(C, C') = 0)$, and (2) $\forall i \neq j \;\; (\forall C \in g_i \;\forall C'' \in g_j \;\; I_3(C, C'') = 1)$. We say that G has *type-3 inconsistency* if $k > 1$ and let $I_3(G) = k$ denote the type-3 inconsistency of G.

The type-3 inconsistencies capture another kind of differences in code clones that may be introduced by modifying identifiers (including names of variables, functions, types, etc.), which is a common practice during copying and pasting code. Often, not all

Code 1 (missing necessary checks in the shaded part)

File: org.eclipse.debug.ui/ui/org/eclipse/debug/ui/memory/AbstractTableRendering.java

```
int colCnt = fTableViewer.getTable().getColumnCount();
TableItem item = fTableViewer.getTable().getItem(0) ;
for (int i=0; i<colCnt; i++)
{
 Point start = new Point(item.getBounds(i).x, ......
 start = fTableViewer.getTable().toDisplay(start);
 ......
 if (start.x < point.x && end.x > point.x)
   return i;
}
```

Code 2

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

```
TableItem item = null;
for (int i=0; i<fTableViewer.getTable().getItemCount(); i++)
     item = ......
if (item != null)
{
 for (int i=0; i<colCnt; i++)
 {
   Point start = new Point(item.getBounds(i).x, ......
   start = fTableViewer.getTable().toDisplay(start);
   ......
   if (start.x < point.x && end.x > point.x)
     return i;
 }
}
```

Figure 3: Sample type-1 inconsistency and bug.

identifiers in clones are modified; occasionally some identifiers that should be changed are left unchanged, and some that should not be changed are changed. These cases may lead to different numbers of unique identifiers in the clones and thus indicate likely bugs. For example, in the third pair of code in Figure 1, "Code 2" is similar to "Code 1," but it has seven unique identifiers (excluding keywords and punctuations), while "Code 1" only has six. In fact, it was confirmed by the GCC developers that the "extra" identifier (`r_stride`) should have been `l_stride` instead.

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

2.3 Classification of Inconsistencies

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

First, the clone groups can be sorted based on their types of inconsistencies so that developers can have an order to decide which clones to inspect first for most likely bugs.

Definition 2.6 (Inconsistency Rank). Given a clone group G, the *inconsistency rank* of the group, denoted by $\text{RANK}(G)$, is a 4-tuple $\langle |G|, I_1(G), I_2(G), I_3(G)\rangle$, where $|G|$ is the number of clones in G.

Given two clone groups G_1 and G_2 and their associated ranks $\text{RANK}(G_i) = \langle |G_i|, I_1(G_i), I_2(G_i), I_3(G_i)\rangle$ for $i \in \{1,2\}$, the order between G_1 and G_2 is given by the lexicographical order between $\text{RANK}(G_1)$ and $\text{RANK}(G_2)$, *i.e.*:

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

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

In addition, based on our experience, type-1 inconsistencies may be further classified into several subtypes, and different subtypes have different likelihoods to be bugs. Such a type-refinement can further help the classification of clone groups and reduce false positives (Section 2.4).

Definition 2.7 (Inconsistency Subtypes). Given a clone pair F_1 and F_2 and their contexts C_1 and C_2, the kinds of C_1 and C_2 (in terms of language constructs) can be one of *switch*, *if*, *loop*, *function-definition* (or *fundef*), and *program* (or *prog*). The subtype of F_1 and F_2, written $I_S(F_1, F_2)$, is defined based on the kinds of C_1 and C_2:

Subtype-1:	$I_S(F_1,F_2) = 1$,	if $\text{KIND}(C_1) = (\textit{fundef} \mid \textit{prog}) \bigwedge \text{KIND}(C_2) = (\textit{fundef} \mid \textit{prog})$
Subtype-2:	$I_S(F_1,F_2) = 2$,	if $\text{KIND}(C_1) = (\textit{fundef} \mid \textit{prog}) \bigwedge \text{KIND}(C_2) = \textit{loop}$
Subtype-3:	$I_S(F_1,F_2) = 4$,	if $\text{KIND}(C_1) = \textit{loop} \bigwedge \text{KIND}(C_2) = (\textit{switch} \mid \textit{if})$
Subtype-4:	$I_S(F_1,F_2) = 8$,	if $\text{KIND}(C_1) = \text{KIND}(C_2) = \textit{loop}$
Subtype-5:	$I_S(F_1,F_2) = 16$,	if $\text{KIND}(C_1) = (\textit{switch} \mid \textit{if}) \bigwedge \text{KIND}(C_2) = (\textit{switch} \mid \textit{if})$
Subtype-6:	$I_S(F_1,F_2) = 32$,	if $\text{KIND}(C_1) = (\textit{switch} \mid \textit{if}) \bigwedge \text{KIND}(C_2) = (\textit{fundef} \mid \textit{prog})$

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

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

The subtypes capture our intuitions on the relations between context inconsistencies and latent bugs: (1) Subtype-6 may indicate a missing conditional check or a redundant check; (2) Subtype-5 and subtype-4 are actually type-1 consistent, but their conditional predicates within different contexts may help refine possible type-2 inconsistencies (*i.e.*, different conditional predicates); (3) Subtype-3 and subtype-2 may indicate that a substantial semantic change is intended among the clones and the code may be less likely a bug; and (4) Subtype-1 may indicate that the clones and their contexts have too few differences to introduce a bug.

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

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

2.4 Filtering Heuristics

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

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

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

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

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

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

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

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

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

`e1+e2` $\equiv$ `e2+e1`: we treat `e1+e2` the same as `e2+e1` because addition is commutative (similarly for other commutative operators, such as `*`, `||`, and `&&`).

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

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

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

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

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

Filter 4: Instead of using the exact tree matching algorithm (Section 2.2) to compute type-2 inconsistencies, use more approximate measures, such as tree editing distances or Euclidean distances [17], to allow small differences in contextual conditions to further prune type-2 inconsistencies.

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

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

After filtering, the remaining clone groups can be inspected for actual bugs. We will show that the estimated amount of inspected code is small w.r.t. the sizes of the original programs. For example, we manually examined less than 12000 lines of code in the Linux kernel, which are collectively about 0.2% of the total 5.6 million lines, to find 57 bugs and programming style issues (*cf.* Section 3). Considering that the maintenance of duplicated code is still mostly manual and little work has been done on finding clone-related bugs, the code inspection burden of our approach is light and worthwhile, especially when compared to manual audits of the entire code base.

We note that it is possible that our filters may prune certain buggy inconsistencies. This is a common trade-off one needs to make: less code inspection burden versus finding more bugs. Section 3 will present results to show that the filters perform well in terms of reducing false positives with few false negatives.

3. EMPIRICAL EVALUATION

3.1 Implementation

Our bug detection algorithm works on top of a clone detection tool. In our implementation, we use Deckard—a scalable, language-independent, tree-based clone detection tool [17]—to detect code clones as input to our bug detection algorithm.

Deckard is based on a novel characterization of subtrees with numerical vectors in the Euclidean space $\mathbb{R}^n$ and a probabilistic clustering algorithm to efficiently cluster these vectors w.r.t. the Euclidean distance. Given a program in a certain language, Deckard (1) generates a parser from a formal syntax grammar for the language, (2) uses the parser to translate sources files into parse trees, (3) produces a set of vectors that capture syntactic information of the trees, (4) clusters the vectors based on the locality-sensitive hashing algorithm [9], and (5) post-processes the vector clusters to generate reports of code clones. Deckard's language-independence and its scalability and accuracy for clone detection make it a good choice for our purpose. It is also worth mentioning that our bug detection algorithm is general and can be applied with other clone detection techniques [2,3,6,7,18,21,23,25]. Although those techniques have algorithmic and parametric differences from Deckard, we do not anticipate any difficulty in using them in our algorithm.

On the other hand, quality and quantity of detected clones clearly impact the effectiveness of our approach. Deckard has three main parameters that may affect the number and quality of its detected clones [17]. The first one is the *similarity* between two pieces of code for them to be considered clones. It ranges from 0.0 to 1.0; the larger the similarity is, the less difference is tolerated among clones,[2] and less clones may be reported. The second parameter is the *minimum token number* for a piece of code to be included. The larger the minimum token number is, the less clones may be reported. The third parameter, *stride*, mainly controls the minimum spatial distance (in terms of tokens in source files) between two clones. The smaller its value is, the more clones may be reported. Smaller strides may also produce more overlapping clones, and the post-processing phase in Deckard may take more time to prune overlapping segments. If stride is set to ∞, only non-overlapping and syntactically complete pieces of code (*e.g.*, a complete `if` statement or a complete `for` statement) are considered for clones.

3.2 Experimental Setup

We now describe the setup for our empirical evaluation. First, for most of our evaluation, we set Deckard's similarity to 1.0, minimum token number to 50, and stride to ∞. These correspond to standard choices in other clone detection tools, and we want to focus on evaluating the bug detection aspects of our approach. We note that the numbers of false positives and negatives may vary with different parameter settings. In Section 3.4, we will evalu-

[2]Certain language elements, such as identifier names, are treated as the same syntactic element although they may be different lexically.

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

Table 1: Characteristics of subject programs.

ate the impact of different choices of Deckard's parameters on the effectiveness of our approach for bug detection.

Second, we choose well-known large open source projects, such as the Linux kernel and Eclipse, as the subjects in our evaluation.[3] These projects are written in different programming languages, C and Java, which can help us evaluate the generality and language-independence of our approach. Table 1 shows some basic statistics on the projects, including their lines of code and numbers of source files. Table 1 also shows clone-related metrics. For each project, it lists the number of clone groups detected by Deckard, the total number of lines of cloned code, and Deckard's time on clone detection. Thus, the 358331 lines of clones in the 7852 clone groups in the Linux kernel and the 70455 lines of clones in the 2246 clone groups in Eclipse form the main code base where we search for bugs in our following experiments.

Finally, our experiments are mainly performed on a machine with a 3GHz Intel Xeon CPU, 8GB of memory, and Fedora Core 5.

3.3 Detection of Inconsistencies and Bugs

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

Table 2 shows how many inconsistencies and bugs we found in the subject programs. For each clone group reported by Deckard, its inconsistency rank and subtype were calculated (*cf.* Section 2.2), and we counted the number of clone groups of each type of inconsistencies (Columns "# Type-i Inc.") and the total number of groups reported as potential bugs (Column "Total # Inc."). We use the number of lines of code (Column "Est. of LoC for Inspc.") in all the groups, including their contexts, to estimate the amount of code that we need to inspect for actual bugs. Such numbers may help readers to understand better the amount of manual effort to inspect the inconsistent clone groups. The amount of code ranges from 0.2% to 0.7% of the original programs, or from 3.2% to 16.2% of the clones. We believe the manual effort can be justified by the large number of detected bugs.

The numbers of actual bugs revealed by each type of inconsistencies are shown in Columns "# Type-i Bugs." The total numbers of bugs, programming style issues, and suspicious clones are also shown in Columns "Total # Bugs," "# Style Issues," and "# Suspects" respectively. For each remaining clone group after filtering, we manually inspected it to check whether it points to a real bug. We made such decisions based on our knowledge of the code: (1) if we have high confidence that an inconsistency causes inappropriate behavior in any clone of the group, we classified it as a bug; (2) if we have high confidence that an inconsistency has no effect on the intended behavior of the clones, we classified it as a false positive; (3) if we believe the clones are behaviorally correct but the code has redundancies or is unnecessarily complicated or confusing, we classified it as a programming style issue; and (4) if we are uncertain about an inconsistency or it takes us too long (more than 30 minutes) to understand the code, we classified it as a suspect. During the examination of a clone group, we may also perform simple data-flow analysis to help understand the code. For most clone groups, the code was fairly easy to understand and the manual inspection took only several minutes each.

[3] We also have preliminary experimental data for GCC 4.0.1, Apache 2.2.0, and JDK 1.5.0, and have found many previously unknown bugs in these projects. However, due to space constraints and the fact that we have not yet inspected all of the bug reports for these projects, we do not report the data here.

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

Table 3: Effects of filters on false positives and negatives. Each row corresponds to different filters (Section 2.4). "None" means no filter was enabled; "All" means all filters were enabled. They are the same data for Table 2.

We were able to find 33 bugs and 9 programming style issues in the Linux kernel and 15 bugs and 13 style issues in Eclipse when all filters were enabled. When fewer filters were enabled, we were able to find more bugs and style issues (Row "Linux w/o filters" and "Eclipse w/o filters" in Table 2). Table 3 also shows the impact of different filters (*cf.* Section 2.4) on bug detection for the Linux kernel. With no filter enabled or all filters enabled, more than 450 false positives were pruned with 15 false negatives. This is a trade-off one has to make between low false positive and negative rates. The bugs exhibit diverse characteristics (Section 3.3.1), and they would be difficult for existing bug detection tools to discover. Considering the relatively light code inspection that is needed, we believe our approach is worthwhile for improving quality of the programs. To date we have received confirmation from developers for two bugs in the Linux kernel and two bugs in Eclipse (and additional ones for GCC and Apache) for the bugs that we have reported. We are continuing analyzing and submitting additional bug reports.

Table 2 also shows the running time of our algorithm (Column "Detection Time"), excluding the time for clone detection and manual inspection. Most of the time was spent on (re-)parsing of clones, the most expensive operation in our approach. As an implementation improvement, we could store parse trees from Deckard to avoid re-parsing, trading space for time.

3.3.1 Breakdown of Bugs and Style Issues

In this section, we categorize the detected bugs and programming style issues in the Linux kernel and Eclipse (Table 4 and 5). In total, there are 41 bugs and 16 style issues in the Linux kernel, and 21 bugs and 17 style issues in Eclipse. We also noticed that the bugs and style issues have diverse characteristics, confirming that many different kinds of bugs can be introduced when developers copy and paste code.

Table 4 lists the main reasons that caused these bugs. Missing necessary conditional checks before using certain data seems to be the most common kind of bugs (Row "ID 1"). Figure 3 shows such an example. Figure 4 shows another error caused by "Wrong function calls." Lines 2674–2721 and lines 2724–2773 are clones, and they have different numbers of unique identifiers. It did not take us long to realize that the call to `pci_bus_write_config_word` on line 2682 should have been `pci_bus_write_config_byte`. Because `pci_bus_write_config_word` takes parameters of type `void *`, the type checker did not catch the mismatch between the type of `temp_byte` and the expected type by the function. At a coarser granularity, most bugs caused by "Wrong function calls," "Wrong variables," "Wrong data fields," and "Wrong macros" may be classified as "Wrong identifiers." The fact that many bugs fall into this category confirms that copying and pasting code often requires identifier renaming, which can be error-prone.

Table 5 shows the kinds of style issues found by our approach. Although some code with style issues may be deliberate, such as for debugging, for code obfuscation, for an experimental or immature feature, or as dummy code, we believe that code with the style issues listed in Table 5 is generally confusing, results in less optimized code, and reduces program readability and maintainability, and it should be avoided as much as possible.

Program	Detection Time (sec)	# Clone Groups	# Type-1 Inc.	# Type-1 Bugs	# Type-2 Inc.	# Type-2 Bugs	# Type-3 Inc.	# Type-3 Bugs	Total # Inc.	Total # Bugs	# Sus-pects	# Style Issues	# False Positives	Est. of LoC for Inspc.
Linux w/ all bug filters	387	7852	115	10	350	25	69	12	396	33	69	9	285	11258
Linux w/o filters	355	7852	177	11	527	29	388	15	881	41	85	16	739	37430
Eclipse w/ all bug filters	127	2246	146	2	249	13	26	2	265	15	42	13	195	6096
Eclipse w/o filters	125	2246	224	4	390	17	91	4	461	21	50	17	373	11536

Table 2: Numbers of inconsistencies and bugs reported when all or no filters (Section 2.4) were enabled.

Code 1 (wrong function call)

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

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

Code 2

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

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

Figure 4: Bug example: a wrong function call.

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

Table 4: Categories of detected bugs.

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

Table 5: Categories of detected style issues.

Here, we only give an example for "Less optimized code" in Figure 5. "Code 1" and "Code 2" were reported as clones, but have different context conditions. One can see that `newWidth` in "Code 1" is calculated more times than necessary (line 592), while "Code 2" is optimized to calculate `width` only once (line 680). Compilers may not be able to perform the optimization automatically because `getClientArea()` is fairly complicated and the compiler may not be able to infer that `newWidth` is a constant.

Some of the bugs and style issues can be detected by existing techniques. For example, missing a `NULL` check (*e.g.*, Figure 3) can be revealed by data flow analyses. However, many bugs may involve programming logic errors, such as inappropriate conditions (*e.g.*, # 1 in Figure 1) and inappropriate scoping (*e.g.*, # 2 in Figure 1), and are difficult to discover without specifications. Section 3.4.2 discusses further how our approach and existing techniques may complement each other.

We also believe that the categories of clone-related bugs and style issues can be useful in two aspects: (1) They can help developers to understand better possible reasons that cause clone-related errors and consciously prevent them from happening again in the future; (2) Automated tools may be implemented to check code clones against each of such categories for code validation.

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

Table 6: Category of inconsistencies that cause false positives.

3.3.2 Breakdown of False Positives

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

Table 6 lists several reasons that are responsible for most false positives in our experiments. Basically, many differences among clones legitimately exist because they are intended to behave differently, such as drivers for devices with slightly different features, and exception handling code for different types of exceptions. Any such intended behavioral differences may cause a false positive in our approach because (1) our current definitions for contexts and inconsistencies do not consider program behavior; (2) all our filters are mainly syntax-based; and (3) Deckard, the clone detection tool used in our approach, is also syntax-based and may report semantically different but syntactically similar code as clones. All of the reasons listed in Table 6 concern program semantics (*e.g.*, types, data and control dependencies) and their intended behavior. It would be interesting to extend the idea of context-based inconsistency and bug detection to semantic-based clones and incorporate semantic information into the definitions of contexts and inconsistencies and the filters to detect bugs more accurately (discussed further in Sections 3.4.3 and 3.4.4).

3.3.3 Comparison with CP-Miner

CP-Miner [25] is a token-based clone detection tool for C. To our knowledge, it is the only existing tool that looks for bugs directly in

Code 1 (less optimized)

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

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

Code 2

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

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

Figure 5: An example of programming style issues: less optimized code.

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

Table 7: Comparison with CP-Miner (Linux kernel 2.6.19).

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

Table 8: Potential effects of different clone detection parameters on false positives and negatives with all filters enabled.

cloned code. In this section, we compare CP-Miner's effectiveness with ours on the Linux kernel. Section 4 will discuss other related bug detection techniques.

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

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

For example, in the third clone pair in Figure 1, let Code 1 be F_1 and Code 2 be F_2, then $\mathit{UnchangedRatio}(\texttt{X1}) = \frac{0}{2} = 0$ because both instances of `X1` have been changed to `X2`. Similarly, we have $\mathit{UnchangedRatio}(\texttt{l\_stride}) = \frac{1}{2}$ and $\mathit{UnchangedRatio}(\texttt{value}) = 1$. Similar to our type-3 inconsistencies, *UnchangedRatio* is used to measure whether programmers change identifiers consistently when they copy and paste code. A non-zero or non-one value for *UnchangedRatio* may indicate inconsistent changes of the identifiers and reveal a potential bug. *UnchangedRatio* is a finer-grained metric than our type-3 inconsistencies, and if a clone pair has type-3 inconsistency, it must have some identifier with a non-zero value for its *UnchangedRatio*, which means CP-Miner may generate more reports than ours and we may miss certain bugs. On the other hand, our type-3 inconsistencies are more efficient to calculate and report fewer false positives.

Table 7 shows our experiments on the Linux kernel 2.6.19, using 50 as the minimum token number and 1.0 as the similarity for both CP-Miner and Deckard. We also set the stride parameter in Deckard to ∞. Deckard reported fewer clones (Column "# Cloned LoC") in slightly longer time (Column "Total Run Time"), thus the initial code base for reporting bugs is smaller for our approach.[4] However, our approach still found more bugs and style issues (Column "# True Pos.")[5] because we look for inconsistencies not only within clones, but also in the contexts which are beyond the clones. We also achieved a much lower false positive rate. All reports (Column "# Positives") generated by CP-Miner and our approach were manually inspected by us. Among the 251 reports from CP-Miner, 55 cases were classified as suspects.

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

3.4 Discussion

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

3.4.1 Which clones to choose from?

Our approach works on code clones detected by Deckard [17], a tree-based clone detection tool. The set of clones may vary when we use different parameters for Deckard, and the bug reports from our approach may also vary. Table 8 shows such effects by varying Deckard's parameters.

Recall from Section 3.1 that the main parameters for Deckard are *similarity*, *minimum token number*, and *stride*. We experimented with different similarities (the first segment of Table 8, by setting the minimum token number to 50 and the stride to ∞), different minimum token numbers (the second segment of Table 8, by setting the similarity to 1.0 and the stride to ∞), and different strides (the third segment of Table 8, by setting the similarity to 1.0 and the minimum token number to 50) on the Linux kernel 2.6.19.

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

3.4.2 Why not just existing bug detection techniques?

Many static and dynamic analysis techniques, such as ESC/-Java [13] and Valgrind [27], exist for bug detection. Static analyses

[4]These results do not imply Deckard performs worse than CP-Miner in general. With different parameter settings, Deckard can detect more clones than CP-Miner in the same amount of time [17].

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

are usually sound—they do not miss bugs with the property that they are looking for. Dynamic analyses are usually accurate—they do not report false positives. However, such techniques usually need to analyze all code in a program for bugs because they do not know in general *where* to analyze, and thus may not be able to scale to programs with millions of lines of code. Also, they usually require certain property specifications so that they can know *what* kinds of bugs to target at, and thus their bug finding capabilities are limited by available specifications.

Compared with those techniques, our approach has mainly two advantages: (1) it effectively reduces the amount of code which requires analysis for bugs; (2) it can hint at possible properties of latent bugs for more specific analyses through the discovered inconsistencies. For example, when the type-3 inconsistency in the third pair of code snippets in Figure 1 was discovered, a simple difference analysis of the data and control dependencies of the two snippets revealed that there is a missing data dependency between `r_stride` and the `if` condition. Then we knew that the latent bug could either be a missing `NULL` check on `r_stride` or a wrong use of `r_stride`. Such advantages can help guide the existing techniques on *where* and *what* to analyze and make them more scalable. In fact, many bugs we found are difficult to be discovered by any single existing technique. We believe that our approach complements well the existing techniques. Conversely, incorporating existing analysis techniques into our approach can provide semantic information to help reduce more false positives and improve the usability of our approach. The following section elaborates on this.

3.4.3 How to reduce false positives further?

Currently, when a clone group is reported as a possible bug, we inspect it in the following way: (1) locate the clones in the original source code and find the actual differences among clones based on their inconsistency ranks and subtypes; (2) inspect the clones and their contexts to look for any hints, such as comments and data dependencies, which can explain the differences; (3) perform manual data-flow analysis to help understand the code whenever necessary.

Many steps in the inspection process can be automated and may help to prune false positives without human intervention. On one hand, we often asked ourselves common questions, such as "where the variable is defined," "whether the return value of this function can be null," and "whether this conditional predicate can ever be false," during code inspection. Most of such semantic-related questions can be easily answered by many program analyses and theorem proving techniques, and help to decide whether an inconsistency is legitimate. As for the purpose of filtering, such techniques do not need to be accurate as long as they can answer the questions with low false negative rates. On the other hand, the inconsistencies among clones can provide hints at what questions to ask. As a simple example, there is a missing `NULL` check for variable `item` in "Code 1" in Figure 3. Based on the difference, it was obvious to ask whether `item` could ever be `NULL` to decide whether the missed check is an actual bug. Generalizing such question-generation schemes and integrating them with other techniques will be like integrating query generators with answer machines, and it will be interesting to investigate how many more false positives may be pruned by an automated code inspection mechanism provided by such an integration.

As another aspect, our current definitions of contexts and inconsistencies are mainly *syntax*-based and only consider the smallest enclosing control-flow construct of a clone. They have not incorporated any semantics of the clones, and neither do the filters for pruning bug reports. It will be interesting to extend our definitions to *semantic*-based representations of programs, such as program dependency graphs [12], so that semantic information, such as types, data and control dependencies, can be considered to help detect more bugs while pruning more intended inconsistencies. Further, we believe that the basic idea that inconsistencies among clones are indications of bugs can be directly applied to semantic-based code clones [21, 23], which are most robust against code modifications, such as re-ordered statements, non-contiguous code, and redundant code, than syntax-based clones. Such clones, together with syntax-based clones, may naturally exclude syntactically similar but semantically different code and thus introduce fewer false positives in the first place.

3.4.4 Applicability

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

One such situation is when we use smaller similarities to generate clones (Section 3.4.1). When a smaller similarity is used, code with more differences may still be treated as clones, and thus inconsistencies can become more commonly intended. In our experiments, we mainly restricted similarity to 1.0 to avoid clones with too many differences. Although such a restriction may miss certain bugs, we believe that it currently is a reasonable trade-off between low false positive and negative rates. In the future, our inconsistency classification and filtering heuristics can be improved to tolerate inconsistencies which are introduced by smaller similarities so that false positive rates can be kept low.

Another situation is when clones evolve independently and intentionally deviate from each other in certain aspects. For example, drivers for several different models of a display card from a same manufacturer have much code in common, but also have many differences that handle different features in the different models. Such inconsistencies among clones may only be indications of different features instead of bugs. For such cases, simple filtering strategies may not always be enough for reducing false positives because the inconsistencies caused by diverse code features may not be easily described by any specific filtering patterns. Certain *inconsistency specifications* from developers, indicating what kinds of differences are intended, can be helpful for reducing false positives. Alternatively, we may apply specification mining techniques, such as [1, 22], to infer such inconsistency specifications first, and then apply our approach to find unintended inconsistencies only.

4. RELATED WORK

In this section, we discuss closely related work and roughly divide them into two categories: (1) studies on clones, and (2) bug detection in clones.

4.1 Studies on Clones

Many algorithms and tools exist for code clone detection. Based on different characterizations of programs, such techniques can be (1) string-based (usually lines in source files) [2, 3], (2) token-based [18,25], (3) tree-based [6,7,17,29], or (4) semantic-based [21, 23]. Because our approach defines inconsistencies based on program syntax and does not reply on a particular clone detection technique, clones detected by any of these techniques can be used directly in our approach.

Also, many studies [1, 5, 11, 22, 26] aim at finding *programming rules*, *structural clones*, or *specifications* which are higher-level similarities in programs than code fragments. For example, "an allocator a must be followed by a deallocator b" is an example of such high-level similarities. Our approach currently only operates at the code level. It would be interesting to extend our notion of inconsistencies to such higher-level similarities for bug finding.

There are also studies that address the questions of clone coverage and evolution and clone removal, other than bug detection. The goal for clone coverage is to determine what fraction of a program is duplicated code. It was confirmed that a significant amount of duplicated code exists in large code bases. For example, CCFinder [18] reported 29% cloned code in JDK, and CP-Miner [25] reported 22.7% cloned code in the Linux kernel 2.6.6.

The goal of clone evolution is to understand how clones are introduced or removed across different versions of a software. Laguë *et al.* [24] examined six versions of a telecommunication software system and found that a significant number of clones were removed due to refactoring, but the overall number of clones increased due to the faster rate of clone introduction. Kim *et al.* [20] described a study of clone genealogies and found that: (1) many code clones are short-lived, so performing aggressive refactoring may not be worthwhile; and (2) long-lived clones pose great challenges to refactoring because they evolve independently and can deviate significantly from the original copy.

The goal of clone removal is to reduce duplicated code and improve code readability and maintainability. Baxter *et al.* [6] extract and refactor code clones so that programs can be rewritten in more generic forms. Jarzabek *et al.* [15, 16] applied a generative technique based on *composition with adaptation* to eliminate redundancies in programs, and aimed at unifying and maintaining clones at meta-levels. However, Rajapakse *et al.* [28] suggested that unifying clones may not be always desirable because of its impact on system qualities, such as performance.

4.2 Bug Detection in Clones

Studies have also proposed bug detection techniques based on the general observation as ours that inconsistencies can be indications of bugs. CP-Miner [25] and our approach operate at the code level. Engler *et al.* [11] and PR-Miner [26] aimed at detecting violations of programming rules. Ammons *et al.* [1] and Kremenek *et al.* [22] considered the problem in the context of program specifications. Also related is Xie *et al.*'s work [31] on using redundancies in programs, such as idempotent operations, unused values, dead code, un-taken conditional branches, and redundant null-checks, to flag possible errors. Dillig *et al.* [10] used semantic inconsistencies among uses of the same pointer to find null-pointer dereference errors. In a broader sense, our work is also related to the large body of work on bug detection techniques, such as [4, 8, 13, 14, 30]. As discussed in Section 3.4, we believe that our approach complements well these existing techniques.

5. CONCLUSIONS

In this paper, we have proposed a general notion of inconsistencies for code clones and presented an approach to locate clone-related errors by detecting such inconsistencies. We have also performed an extensive evaluation of our approach on large open source projects, including the Linux kernel and Eclipse. We were able to discover many previously unknown bugs and programming style issues in these projects, confirming the hypothesis that code cloning can be error-prone. We also found that, due to the diverse characteristics of the clone-related bugs, they cannot be easily discovered with any single existing program analysis technique, and thus our proposed approach complements well these existing techniques for bug detection.

6. REFERENCES

[1] G. Ammons, R. Bodik, and J. R. Larus. Mining specification. In *Symposium on Principles of Programming Languages (POPL)*, 2002.

[2] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Working Conference on Reverse Engineering (WCRE)*, pages 86–95, 1995.

[3] B. S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM Journal on Computing (SICOMP)*, 26(5):1343–1362, 1997.

[4] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Symposium on Principles of Programming Languages (POPL)*, pages 1–3, 2002.

[5] H. A. Basit and S. Jarzabek. Detecting higher-level similarity patterns in programs. In *ESEC/FSE*, pages 156–165, 2005.

[6] I. D. Baxter, C. Pidgeon, and M. Mehlich. DMS®: Program transformations for practical scalable software evolution. In *ICSE*, pages 625–634, 2004.

[7] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM*, pages 368–377, 1998.

[8] M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 57–68, 2002.

[9] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Symposium on Computational Geometry (SoCG)*, pages 253–262, 2004.

[10] I. Dillig, T. Dillig, and A. Aiken. Static error detection using semantic inconsistency inference. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 435–445, 2007.

[11] D. R. Engler, D. Y. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Symposium on Operating Systems Principles (SOSP)*, pages 57–72, 2001.

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

[13] C. Flanagan, K. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245, 2002.

[14] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages (POPL)*, pages 58–70, 2002.

[15] S. Jarzabek, P. Bassett, H. Zhang, and W. Zhang. XVCL: XML-based variant configuration language. In *ICSE*, pages 810–811, 2003.

[16] S. Jarzabek and S. Li. Eliminating redundancies with a "composition with adaptation" meta-programming technique. In *ESEC/FSE*, pages 237–246, 2003.

[17] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *ICSE*, pages 96–105, 2007.

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

[19] C. Kapser and M. W. Godfrey. "cloning considered harmful" considered harmful. In *Working Conference on Reverse Engineering (WCRE)*, pages 19–28, 2006.

[20] M. Kim, V. Sazawal, and D. Notkin. An empirical study of code clone genealogies. In *ESEC/FSE*, pages 187–196, 2005.

[21] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *International Static Analysis Symposium (SAS)*, pages 40–56, 2001.

[22] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: Inferring the specification within. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 161–176, 2006.

[23] J. Krinke. Identifying similar code with program dependence graphs. In *Working Conference on Reverse Engineering (WCRE)*, pages 301–309, 2001.

[24] B. Laguë, D. Proulx, J. Mayrand, E. Merlo, and J. P. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *ICSM*, pages 314–321, 1997.

[25] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 289–302, 2004.

[26] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE*, pages 306–315, 2005.

[27] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 89–100, 2007.

[28] D. C. Rajapakse and S. Jarzabek. Using server pages to unify clones in web applications: A trade-off analysis. In *ICSE*, pages 116–126, 2007.

[29] V. Wahler, D. Seipel, J. W. von Gudenberg, and G. Fischer. Clone detection in source code by frequent itemset techniques. In *International Workshop on Source Code Analysis and Manipulation*, pages 128–135, 2004.

[30] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *Symposium on Principles of Programming Languages (POPL)*, pages 351–363, 2005.

[31] Y. Xie and D. R. Engler. Using redundancies to find errors. In *FSE*, pages 51–60, 2002.