Scalable and Systematic Detection of Buggy Inconsistencies in Source Code

Mark Gabel[1] Junfeng Yang[2*] Yuan Yu[3] Moises Goldszmidt[3] Zhendong Su[1†]

[1] University of California, Davis {mggabel, su}@ucdavis.edu
[2] Columbia University junfeng@cs.columbia.edu
[3] Microsoft Research, Silicon Valley {yuanbyu, moises}@microsoft.com

Abstract

Software developers often duplicate source code to replicate functionality. This practice can hinder the maintenance of a software project: bugs may arise when two identical code segments are edited inconsistently. This paper presents DejaVu, a highly scalable system for detecting these general syntactic inconsistency bugs. DejaVu operates in two phases. Given a target code base, a parallel *inconsistent clone analysis* first enumerates all groups of source code fragments that are similar but not identical. Next, an extensible *buggy change analysis* framework refines these results, separating each group of inconsistent fragments into a fine-grained set of inconsistent changes and classifying each as benign or buggy. On a 75+ million line pre-production commercial code base, DejaVu executed in under five hours and produced a report of over 8,000 potential bugs. Our analysis of a sizable random sample suggests with high likelihood that at this report contains at least 2,000 true bugs and 1,000 code smells. These bugs draw from a diverse class of software defects and are often simple to correct: syntactic inconsistencies both indicate problems and suggest solutions.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging; D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement—restructuring, reverse engineering, and reengineering

General Terms Languages, Reliability, Algorithms, Experimentation

* Junfeng Yang was supported in part by NSF Grant Nos. CNS-1012633 and CNS-0905246.

† Zhendong Su was supported in part by NSF CAREER Grant No. 0546844, NSF CyberTrust Grant No. 0627749, NSF CCF Grant No. 0702622, NSF TC Grant No. 0917392, and US Air Force Grant No. FA9550-07-1-0532.

OOPSLA/SPLASH'10, October 17–21, 2010, Reno/Tahoe, Nevada, USA.

1. Introduction

Software developers often duplicate source code to replicate functionality. This duplication may be performed simply for convenience, or it may represent more thoughtful solutions to difficult engineering problems. In either case, copied code can be difficult to maintain. Identical code is often expected to evolve in sync, with feature additions and bug fixes applied consistently across copies. In practice, this can be quite challenging. For instance, copied code may lack annotations that point a would-be editor to all other identical code segments, so consistent updates become dependent on a developer's memory and/or diligence. Large systems with multiple authors further complicate the situation. For example, a new developer of an operating system may use a copy of an existing device driver as a template—without informing the original author. Lacking communication, a bug in the shared portions of the code would not likely be fixed consistently.

Researchers have recognized these difficulties and have studied the duplicate code problem extensively. Examples of technical work include the formalization of the notion of similar code fragments, referred to as *code clones*, and the development of efficient algorithms for enumerating all copied code in a software system, implemented as *clone detection tools* [2, 3, 15, 18, 21]. Other, more empirical work has studied the incidence of code cloning both qualitatively [19, 20] and quantitatively [22, 25].

Most of the research on code clones advocates some form of proactive solution to the clone maintenance problem; that is, the research introduces solutions that aim to *prevent* the introduction of bugs to copied code. For example, much of clone-related research makes the explicit or tacit suggestion that cloning code should be avoided and that cloned code itself should be factored out into a single, parameterized copy whenever possible: Baker's seminal paper on the topic [2] discusses the total reduction of a program's size given a complete factorization of *all* cloned code, putting the code in a certain 'normal form.' Other research provides support for managing existing code clones, providing tools to track copied portions of code [6, 23] and update them consistently [27].

Comparatively few projects have provided support for some form of retroactive *detection* and repair of clone-related bugs, the two notable examples being Li *et al.*'s CP-Miner [21] and Jiang *et al.*'s context-based clone bug detector [16]. Though effective, these bug finding tools have fundamental limitations.

Recall: Quantity and Variety of Bugs Both of these tools focus on a specific class of clone-related bugs we call *copy-time* bugs. These bugs are introduced *when* code is copied: they are a result of improperly adapting copied code into its new environment. Each tool is further restricted to a specific subclass of copy-time bugs: CP-Miner finds instances of copied code with inconsistently renamed identifiers, while Jiang *et al.*'s tool extends that approach by additionally locating code clones with conflicting surrounding contexts (*e.g.* an `if` statement versus a standard code block), which may indicate that the preconditions for execution of the code were not understood when copying.

These tools do not sufficiently address the general—and likely much more common—case of bugs in inconsistent *edits* to clones, which usually occur after the initial copy. The editing of code clones, be it bug fixes other evolutionary edits, involves the insertion, deletion, and/or replacement of arbitrary code. As these specialized tools are based on an analysis of *exact* code clones, only the rare instances of inconsistent edits that fit their respective models will be detected—the majority will likely be missed.

Precision: Quality of Bug Reports Despite their focus on specific classes of bugs, existing tools suffer from a large number of false bug reports: both Li *et al.* and Jiang *et al.* report the precision of their tools as approximately 10%, *i.e.*, 90% of their output consists of false alarms. While one must grant a certain amount of latitude to tools based solely on the analysis of syntax, the signal to noise ratio of *these* tools is likely at or below many developers' thresholds for tolerance.

Scalability While existing clone *bug* detection tools are sufficiently scalable to moderately large (*e.g.* several million lines of code) software systems, none have been demonstrated to scale to very large projects (*e.g.* tens to hundreds of millions of lines). This level of scale would hardly be a purely academic exercise: commercial code bases can grow to this size for a single large project.

DejaVu In this paper, we address all of these limitations with DejaVu, a highly scalable and precise system for a detecting buggy inconsistencies in source code. DejaVu implements a static source code analysis that captures both the class of copy-time clone bugs—those found in poorly-adapted copies of source code—and the more general class of inconsistent edits to clones. Our work is inspired by and builds on previous work on the detection of *near-miss* clones in software: a limited number of existing clone *detection* tools [15, 21] are able to boost recall (the amount of copied code found) by allowing for minor differences in clones, *e.g.* clones differing by a few tokens. At a high level, DejaVu functions quite similarly: we also search for 'near miss' clones, but we treat the inconsistencies as potential *bugs*. Though conceptually straightforward, this approach presents several key challenges:

- Focusing on 'near' misses is too restrictive. In order to detect a broad class of inconsistent edits, we must push the concept of 'near miss' clones to its logical limit and allow for highly divergent clones. Discrepancies between pairs of code fragments may exceed a 'few' tokens: inconsistent edits may involve several entire code blocks, and the changes need not necessarily be contiguous.
- Precision is paramount. Relaxing the definition of 'similarity' creates a natural precision problem by potentially admitting many spurious, non-copied 'clones.' There is a more subtle issue as well: a pair of inconsistent code fragments makes for a rather crude bug report, as any given pair of fragments may contain several classes of changes: some will be intentional and adaptive, others will be unintentional but innocuous, and only a fraction of them, if any, will be buggy.
- No clone detection tool capable of finding divergent clones has been demonstrated to scale beyond inputs of a few to several million lines of code. DejaVu's primary target is a 75+ million line pre-production commercial code base that is anecdotally believed to contain a significant amount of copied code.

DejaVu contains two principal components. While each is separable as a unique contribution, we believe our most important contribution is our demonstration of their effectiveness as a practical, integrated system.

Large-scale Inconsistent Clone Analysis We implement an inconsistent clone detection tool for C and C++ programs. It is both parallel and distributed, taking advantage of both local and network computing power. Using four quad-core systems, DejaVu's clone detection component is able to enumerate all clones in a 75+ million line system—using a quite liberal definition of similarity—in just a few hours. The system is based on a significant reworking and an independent implementation of the current state-of-the-art algorithm, Jiang *et al.*'s DECKARD [15]. Our extensions, among other things, add the aforementioned parallelization and a guarantee of *maximality* of the returned clone sets: we report the largest possible clones without reporting any redundant constituent components, greatly improving the precision and completeness of the bug reports.

Buggy Change Analysis Framework Our clone analysis is tuned for extreme recall; at its most aggressive settings, the raw clones are barely usable on their own. In spite of this, our system maintains a high degree of precision through our *buggy change analysis framework*. This framework—which can be 'bolted on' to any clone detection tool—uses a combination of flexible lexical analysis, sequence differencing and similarity algorithms, and the extraction of other

```
1  int    status = OK;
2  int    *resdi = NULL;
3  int    res = NULL;
4  AnsiString   resourcePath;
5  RESOURCEINFO  resinfo;
6  FRE     fre;
7  bool    isReady = false;
8
9  Check(resourcePath.Set(paramPath, CP_UTF8));
10 CheckTrue(res = RsrcOpen(resourcePath, BINARY | RDONLY
       | SEQUENTIAL, 0), E_FAIL);
11
12 // Bug Ver 17, Bug #12743: We no longer need to wrap
       malloc and free.
13 CheckTrue(resdi = RsrcCreate(malloc, free, RsrcOpen,
       RsrcRead, RsrcWrite, RsrcClose, RsrcSeek, UNKNWN,
        &fre), E_FAIL);
14 isReady = RsrcIsReady(resdi, res, &resinfo);
15
16 Error:
17 if (resdi)
18   RsrcCleanup(resdi);
19 if (res)
20   RsrcClose(res);
21 return isReady;
```

```
1  int    status = OK;
2  int    *resdi = NULL;
3  int    res = -1; // Invalid resource handle
4  AnsiString   resourcePath;
5  RESOURCEINFO  resinfo;
6  FRE     fre;
7  bool    isReady = false;
8
9  Check(resourcePath.Set(paramPath, CP_UTF8));
10 res = RsrcOpen(resourcePath, BINARY | RDONLY |
       SEQUENTIAL, 0);
11
12 // Bug Ver 19, Bug #256: Invalid return; crashes.
13 CheckTrue(res != -1, E_FAIL);
14 if (res == -1)
15   // it is safe to return directly here.
16   return false;
17
18 // Bug Ver 17, Bug #12743: We no longer need to wrap
       malloc and free.
19 CheckTrue(resdi = RsrcCreate(malloc, free, RsrcOpen,
       RsrcRead, RsrcWrite, RsrcClose, RsrcSeek, UNKNWN,
        &fre), E_FAIL);
20 isReady = RsrcIsReady(resdi, res, &resinfo);
21
22 Error:
23 if (resdi)
24   RsrcCleanup(resdi);
25 if (res != -1)
26   RsrcClose(res);
27 return isReady;
```

Figure 1: Motivating Example. A clearly annotated example of an inconsistent fix to a pair of copied code segments.

clone-related features to provide a configurable framework for *refining* the inconsistent clones and *isolating* the buggy changes. This step is computationally expensive: executing it in a brute force manner over the entire target code base would be intractable; it is only in linking it with the inconsistent clone analysis—which serves to massively prune the search space—that it becomes efficient.

Our experiments indicate that DejaVu is an effective bug finding system: on our 75+ million line pre-production code base, DejaVu produced 8,103 bug reports. We sampled and verified a random sample of 500 of these reports and found 149 very likely bugs and 109 other code smells. Using standard techniques, we estimate with high confidence that DejaVu has found 2,070–2,760 new bugs in this code base. These bugs are varied in nature, often difficult to detect using other techniques, and, given that the fixes are often embedded in the bug reports, simple to repair.

This paper is organized as follows. In the next section (Section 2), we provide a motivating example and a high-level overview of DejaVu. In the following two sections (Sections 3 and 4) we describe our system's two main components in detail. Section 5 describes our experimental results, and Sections 6 and 7 discuss related work and our plans for continuing this research, respectively.

2. System Overview

Our presentation of DejaVu begins with a motivating example that highlights the advantages of our approach and provides insight into our design decisions. We continue with a high-level architectural view of the system.

2.1 Motivating Example

Consider the code fragments in Figure 1. These fragments are clones[1] that perform the same function: they are part of separate modules that provide object-oriented wrappers around the same system resource. The left fragment contains a critical bug: on line 10, the programmer mistakenly checks the return value of `RsrcOpen` as a Boolean, assuming "true" indicates a successful return. (The `Check` and `CheckTrue` identifiers refer to control flow macros that jump to the `Error:` label, commonly used for cleanup code, if a condition is false.) However, this particular system function returns a negative nonzero value on error, which is incorrectly interpreted as success. This is a real bug that was located by our system. It exhibits several interesting properties:

[1] All "bug" examples in this paper are taken directly from a snapshot of our commercial code base. They have been anonymized, but we have made an effort to guide the anonymization such that the apparent semantics of each code fragment remains clear.

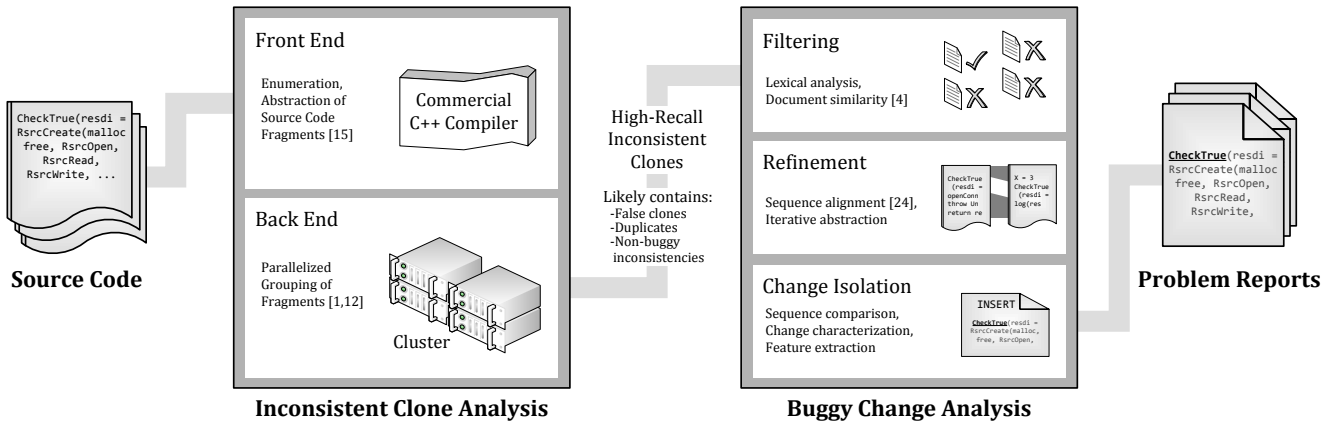


Figure 2: DejaVu system architecture.

- On line 12 of the right fragment, a comment clearly shows that this programming error had actually caused a runtime crash and had been entered into an issue tracking database as a numbered bug. The programmer corrected the condition of the `CheckTrue` call (and added an unnecessary second check) but neglected to propagate the change to the fragment's twin. This is congruent with the intuitive notion of a copy-paste bug.
- The presence of other identical comments (lines 12 and 18 of the left and right fragments, respectively) suggests that the fragments either were at one point edited consistently or were duplicated after the comment was added.
- The fragments exhibit both inter-line and intra-line structural differences, including the insertion of complete statements and the modification of a condition, an initializer, and a function call. DejaVu handles these general differences while still maintaining a high degree of precision.
- The fragments contain both buggy and irrelevant changes. The initializations on line 3 of both fragments are both redundant dead stores and are not relevant to the bug.
- Though not depicted in the figure, these code fragments currently have different "owners" and are separated by several levels in the project directory structure. This pattern of copying code across large "distances" is not uncommon and necessitates DejaVu's global analysis.

This example clearly motivates our technique: a scalable, global static analysis that enumerates divergent copies of otherwise duplicate code and isolates and classifies their essential differences.

2.2 System Architecture

This section provides a high-level overview of each of DejaVu's components in the context of the implemented system; we will later present each component's design and implementation in detail in Sections 3 and 4. DejaVu's architecture appears in Figure 2. At a high level, DejaVu consists of two components: the inconsistent clone detector and the buggy change analysis framework.

Clone Detection 'Front End' The front end of our clone analysis is responsible for reading the input source code and producing a set of *clone candidates*, abstractions of code fragments that can be potentially grouped together as clones. It is implemented within the front end of a commercial C++ compiler and thus easily integrates with a variety of build systems. Aside from simplicity of execution, the integration with build systems also allows incremental clone detection.

Clone Detection 'Back End' After DejaVu produces a set of candidate code fragments, the back end of our clone analysis clusters the candidates to form a set of clones. This step is the dominant computation—especially when we increase the computational cost by tuning the analysis for recall—and it is parallelized on a compute cluster. Though omitted from the Figure 2 for brevity, a single master node coordinates the computation and handles data distribution and aggregation. Each node in our compute cluster is equipped with a multicore processor, which we also fully utilize.

Buggy Change Analysis Framework Upon completion, the clone analysis yields a large, but potentially noisy, set of inconsistent clones. We refine this set and distill the essential buggy changes with an extensible framework. Briefly, our change analysis framework operates in three steps.

1. We first provide a mechanism for applying *filters* to the input clones. In our default implementation, we implement a single main filter: a check based on a document similarity algorithm that ensures (with high probability) that each clone is the product of an *intentional* copy and paste action.
2. Next, we use a sequence differencing algorithm in a series of iterations to *align* the clones and trim their potentially spuriously-matching prologues and epilogues. This focuses our analysis on the essential differences.
3. Using the results of the previous step, we distill the difference between the two clones into a set of fine-grained

changes, which we then categorize and filter based on their characteristics. If essential, likely-buggy changes remain after this step, we output a bug report consisting of a) the pair's source code fragments along with b) a concise 'diff,' highlighting *just* the changes DejaVu finds suspicious.

These steps are potentially quite computationally expensive, and attempting to use this framework in isolation by feeding an entire project through it would be cost-prohibitive. When acting as part of the complete DejaVu system, however, the *clone* analysis handles the scalability problem in stride, transforming a large quadratic-time problem—a similarity search over a large code base—into a much smaller (by orders of magnitude, but still large), linear-time one—the refinement and classification of a set of syntactic inconsistencies.

3. Detecting Inconsistent Clones

Our clone analysis is based on a reworking—and an entirely independent, from scratch implementation—of Jiang *et al.*'s DECKARD algorithm [15]. DECKARD is a general algorithm that operates over tree structures: given a tree model, it scalably enumerates all pairs of similar subtrees and/or subforests, where similarity is defined by tree edit distance. When applied to trees of source code (either concrete or abstract syntax) it functions as a highly effective clone detection algorithm. The algorithm operates in two high level phases. It first enumerates all possible clone candidates and abstracts them into *characteristic vectors*. It then uses *locality sensitive hashing* [12] to scalably group the vectors by similarity and enumerate the similar sets.

This separation of concerns allows a great deal of flexibility. In our case, this design allowed us to readily adapt the algorithm to our requirements: DejaVu implements an entirely new front end, which generates a *complete* set of candidates, allowing us to find *maximal* clones (Section 3.1). We were then able to independently optimize and rework the back end (Section 3.2) to allow for parallel execution, a more principled treatment of inconsistent clones, and several performance optimizations.

3.1 Front End: Candidate Selection and Abstraction

Candidate Selection Our algorithm operates over abstract syntax trees. A natural choice for candidate code fragments are those contained by subtrees, the most relevant of which are function definitions, blocks, and control flow constructs (*e.g.* loops and conditionals). This is a sound first step; DejaVu does include all subtree-delineated candidates. However, this set is rather coarse grained and likely to miss important instances of copying.

Consider again the code fragments in Figure 1. While these fragments comprise complete function definitions, it is conceivable—perhaps even likely—that a third similar code fragment might exist in a form like Figure 3. Note that this code fragment contains at least a portion of the same bug: `res` should be checked against -1 on line 8, not as a

```
1  // ... Several lines of unrelated code ...
2  CheckTrue(resdi = RsrcCreate(malloc, free, RsrcOpen,
       RsrcRead, RsrcWrite, RsrcClose, RsrcSeek, UNKNWN,
        &fre), E_FAIL);
3  isEOF = RsrcIsEOF(resdi, res, &resinfo);
4
5  Error:
6  if (resdi)
7    RsrcCleanup(resdi);
8  if (res)
9    RsrcClose(res);
10 return isEOF;
```

Figure 3: A fragment exhibiting a similar bug.

simple Boolean test. As it is a subsequence of a much larger flat list of statements, this code fragment has no minimal surrounding construct; *i.e.*, it does not represent a subtree. With the use of subtree-only candidates, this bug would likely remain undiscovered.

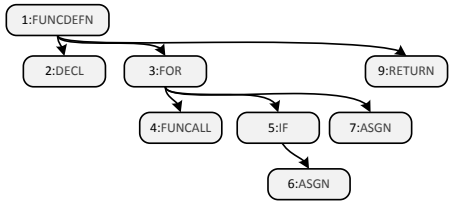
Jiang *et al.* provide a strategy for mitigating this risk in the original implementation of DECKARD [15] by serializing the syntax tree, effectively creating an unstructured token sequence, and using a small (30 to 50 token), fixed-size sliding window to generate additional candidates. For our bugfinding application, this strategy is unsatisfactory. It biases the algorithm toward finding many small clones, possibly with many repeated lines. We require *maximal* clones: around each potentially bug-inducing inconsistency, we must find as much consistent *context* as possible to be able to state with confidence that two code fragments are "nearly identical." Our system's output must also be as free of duplicate bug reports as possible.

Our solution to this problem is a somewhat brute force approach: for each node (equivalently, subtree) in the abstract syntax tree, we generate additional "merged" candidates for all $\binom{n}{2}$ subsequences of the node's children. Put more concretely, for each block, control structure, or function definition, we generate a candidate for each contiguous subsequence of top-level statements.

Example Consider this artificial code snippet:

```
int main() {
    int i = 0;
    for (; i < 10; ++i) {
        cout << "Hello World!";
        if (i >= 10)
            i = 2;
        global = i;
    }
    return 0;
}
```

and its associated (simplified) abstract syntax tree:



Assuming we set a minimum candidate size of two statements, we would first generate candidates for the subtrees rooted at nodes (eq. lines) 1, the enclosing function and 3, the loop. We would then generate "merged" candidates for all subsequences of each node with at least two children. For the enclosing function definition, we would generate the additional candidates consisting of nodes (2 3), (3 9), and (2 3 9). Similarly, we recursively apply the same procedure to the loop (node 3) and generate sequences (4 5), (5 7), and (4 5 7). We are, in effect, enumerating every syntactically valid code fragment that a developer could have copied and pasted.

DejaVu's front end theoretically runs the risk of producing an inordinately large number of candidates: at each "level" of the abstract syntax tree, we are enumerating a quadratic number ($\binom{n}{2} = \frac{n(n-1)}{2}$) of candidates. In practice, this is not a problem: for our 75+ million line code base, we generate fewer than 50 million potential candidates. There are several reasons for this.

- *Minimum Size* We set our minimum candidate size at 6 statements or statement-expressions. Thus, many small blocks are never merged, and there are fewer (but still an asymptotically quadratic number of) subsequences.
- *Canonicalization* We implemented the ability to canonicalize the abstract syntax tree with configurable transformations. At present, DejaVu only performs a single transformation: contiguous sequences of simple declarations, usually found at the top of a given code block, are collapsed into a single logical subtree, rooted with an artificial 'null' node. In our early experiments, considering all subsequences of simple declarations caused a blowup in the number of candidates with little quantitative (*e.g.*, clone coverage metrics) impact on the result set.
- *Configuration of Merging* We have also implemented the ability to disable merging on certain parent node types. As implemented, we only add this restriction to a single case: we do not generate candidates for all possible subsequences of `case`s of a `switch`. Note, though, that we still generate candidates for the subsequences of statements within the blocks delineated by the `case` labels.

Abstraction We have described our enumeration strategy for a *complete* set of clone candidates. Following DECKARD, we abstract each candidate into a *characteristic vector*, to be later clustered by the back end with respect to a similarity measure. Briefly, a characteristic vector is a point in n-dimensional space, where n is the number of tree *node types*.[2] In a given point, the value of each coordinate counts the number of occurrences of a given node type in the clone candidate. In our small example earlier, the following vector characterizes the entire fragment:

$$\langle 1_{\text{FUNCDEFN}}, 1_{\text{DECL}}, 1_{\text{FOR}}, 1_{\text{FUNCALL}}, 1_{\text{IF}}, 2_{\text{ASGN}}, 1_{\text{RETURN}} \rangle$$

[2] Our C++ abstract syntax trees have 124 different node types.

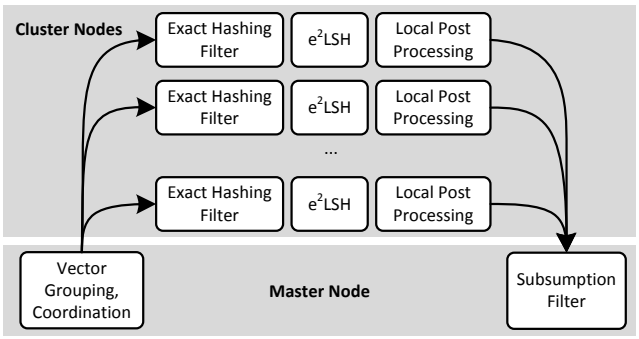


Figure 4: Clone detection back end.

Note that this characterization necessarily involves the full abstraction of all constant values and identifier names. An important property of these vectors is that despite their heavy abstraction, the Euclidean distance between any two points correlates strongly with the tree edit distance between the two clone candidates [28]; thus, clustering with respect to Euclidean distance yields similar clones. For a more detailed presentation of this abstraction, see Jiang *et al.*'s paper [15].

It is important to note that both layers of abstraction—abstract syntax trees and characteristic vectors—are sound: two identical (syntactically complete) code fragments will always correspond to identical abstract syntax trees, and two identical abstract syntax trees will always correspond to identical characteristic vectors. Each layer of abstraction provides us with an opportunity to find more clones (*i.e.*, increase recall) in a more scalable manner (with a reduction in the problem space) at the possible expense of precision.

3.2 Back End: Grouping

The back end of our clone analysis clusters the abstracted clone candidates (*i.e.*, the vectors) into sets of clones. The key enabling technology that allows this to be performed scalably is *locality sensitive hashing* [12]. Briefly, LSH is a highly scalable probabilistic technique for hashing similar items to the same bucket, thus transforming a linear search for similar objects into simple collision detection (which takes constant time). The DECKARD approach involves using an instance of LSH to solve a *near neighbor* problem with the candidate clones, thus grouping similar code fragments. Jiang *et al.* use Andoni's e^2LSH package [1] to implement this solver; we also use Andoni's implementation in binary form.

LSH forms the basis for the scalability of our approach, but several challenges remain in attempting to scale the analysis up to 75+ million lines of code while continuing to produce accurate results that are meaningful for bug detection. Figure 4 displays the interaction of these components, which are described in detail in the following sections.

Parallelization A very large code fragment is never a clone of a very small code fragment, and larger code fragments should tolerate proportionally more edits (inconsistencies) than smaller ones. Jiang *et al.* formalize this notion and pro-

vide a formula for dispatching the vectors into *overlapping subgroups* based on their magnitude, which closely corresponds to code size. Each subgroup has a progressively larger average magnitude and is assigned increasingly more relaxed clustering parameters (*i.e.*, allowing for more inconsistency).

We adapt and implement a similar formula. In addition, we take the novel step of dispatching the subgroups to nodes in a compute cluster and processing them in parallel. For moderately large target projects (around 5 million lines of code), this clustering step completes in around 10 to 20 minutes and does not dominate the overall process. However, for our very large code base, clustering can take tens of hours. Partitioning the work allows a near linear speedup in the number of nodes. With the use of a cluster of four quad core servers, our entire clone analysis completes in a few hours—rather than (up to) several days.

Coordination is handled on a single master node. We schedule the clustering of the subgroups in decreasing order of size to minimize overall latency. In addition, we have implemented basic fault tolerance; the master node detects and retries any failed or incomplete clustering jobs.

Filtering of Identical Vectors As a performance optimization, we preprocess each clustering task by condensing all fragments with identical vectors (*i.e.*, the exact clones) into sets of fragments mapping to a single vector. We implement this in linear time using standard hashing. In practice, this yields a 20 to 30% reduction in the number of vectors that the LSH library must consider. As this optimization is local to a given task (vector group), we distribute this work to the cluster as well.

Local Post-Processing A given set of similar clone candidates is unlikely to be succinct and meaningful without further processing. Consider a pair of exactly duplicate code fragments, surrounded by inconsistent context:

```
1  RSLT result = *gResult;
2  if (result.op == ADJ) {
3    min = result.xMin;
4    max = result.xMax;
5  }
6  else {
7    min = result.yMin;
8    max = result.yMax;
9  }
10 int mid = min/2+max/2;
```

```
1  object->get(&result,0);
2  if (result.op == ADJ) {
3    min = result.xMin;
4    max = result.xMax;
5  }
6  else {
7    min = result.yMin;
8    max = result.yMax;
9  }
10 printf("min=%d\n",min);
```

Lines 2-9 of each fragment are clearly exact clones. However, when allowing for an inconsistency of approximately one statement, lines 2-9 of the first fragment are also duplicates of a) lines 1-9 and 2-10 of the right fragment and b) lines 1-9 and 2-10 of the *same* (left) segment. Intuitively, we see a pair of exact clones, but using this relaxed definition of similarity, the tool incorrectly detects a group of 6 similar code fragments. This highlights a major problem with a very common case: nearly every exact clone can be trivially expanded to form an inexact clone, albeit not in a meaningful way that finds inconsistent edits.

In general, we would like to extract from each group the *most alike*, *non-overlapping* subset of code fragments. We implement a fast, approximate solution to this problem that makes use of each fragment's characteristic vectors: we favor the "most alike" code fragments by prioritizing those whose vectors appear most tightly clustered in the Euclidean space. For a given clone set, we first calculate the geometric *centroid* point of its characteristic vectors. Next, we calculate the distance between the centroid and each individual code fragment's point. After sorting by this distance, ascending, we greedily prune overlapping code fragments. In general, exact clones—if they exist—fall naturally toward the center of the cluster and are listed first. The 'trivial' inconsistent clones are sorted later in the sequence and are pruned due to overlap.

In our experience, this refinement tremendously improved the quality of the results. With ad-hoc random pruning of the overlapping portion of groups, over half (extrapolating from random samples) of the *inconsistent* clones were trivial extensions of exact clones. When using this technique, this proportion dropped to below one percent.

Global Post-Processing: Maximal Clones As a final step, we prune all fully *subsumed* clone groups from the result set. For example, if two functions are complete clones, the subsumed "sub-clones" consist of all pairs of their constituent blocks. These subsumed groups are quite common: because we generate a complete set of candidates, the presence of a larger clone group in the result set implies the existence of many smaller, subsumed clones. We accomplish this pruning in amortized $O(n \log n)$ time by maintaining a priority heap of clone groups, sorted lexicographically (after grouping those with equivalent cardinality) first *ascending* by their members' starting lines and then *descending* by their members' ending lines. Sorting in this way ensures that subsumed groups are adjacent, so pruning them is a linear time operation. To minimize memory overhead, we implemented an immutable string library with internalization. Note that because we generate a complete set of candidates, this pruning phase yields a minimal set of maximal clones—ideal for bug finding.

4. Buggy Change Analysis

DejaVu's buggy change analysis framework takes a large, potentially noisy set of inconsistent *clones* and extracts from it a set of bug reports in the form of specific, likely-buggy inconsistent *changes*. Though this component is an integral part of our system as a whole, it is separable and usable with any clone detection tool, albeit less effectively if provided with fewer potential clones than our own recall-tuned clone analysis is able to detect.

The first stage of this process (Section 4.1) provides an interface for the coarse-grained filtering of clones, which we utilize sparingly. The second stage (Section 4.2) refines each clone pair by performing a series of abstraction and sequence alignment operations. The final, most important

```
if (cd != cdNone) goto Error;
if (cdDEF != cdNone) { cd = cdDEF; goto Error; }

// read data
cd = ReadData(stream, false, &cnt, &data);
if (cd != cdNone) goto Error;

// hand back results.
*pcnt_output = cnt;
*pdata_output = data;
data = NULL;

Error:
if (data != NULL)
   free(&data);
Release(stream);
Release(requestData);
```

```
if (cd != cdNone) goto Error;
if (cdDEF != cdNone) { cd = cdDEF; goto Error; }

// read data types
cd = ReadTypes(stream, &ntypes, &buffer);
if (cd != cdNone) goto Error;

// hand back results.
*pntypes = ntypes;
*pbuffer = buffer;
buffer = NULL;

Error:
if (buffer != NULL)
   free(buffer);
Release(stream);
Release(requestData);
```

(a) A bug: the accidental `free`ing of a stack address. Though they contain several differences, the fragments' textual similarity is > 0.5.

```
if (flags & BORDER)
  obj->OptBorder = true;
if (flags & SHADING)
  obj->OptShading = true;
if (flags & FONT)
  obj->OptFont = true;
if (flags & COLOR)
  obj->OptColor = true;
if (flags & SCROLL)
  obj->OptScroll = true;
```

```
if (flags & RowsBit)
  obj->OptRows = true;
if (flags & LastBit)
  obj->OptLast = true;
if (flags & ColsBit)
  obj->OptCols = true;
if (flags & FirstBit)
  obj->OptFirst = true;
if (!(flags & NoTabs))
  obj->OptTabs = true;
```

(b) A false clone with text similarity < 0.5

```
if ID & ID
  ID -> ID = BOOL
if ID & ID
  ID -> ID = BOOL
if ID & ID
  ID -> ID = BOOL
if ID & ID
  ID -> ID = BOOL
if ID & ID
  ID -> ID = BOOL
```

```
if ID & ID
  ID -> ID = BOOL
if ID & ID
  ID -> ID = BOOL
if ID & ID
  ID -> ID = BOOL
if ID & ID
  ID -> ID = BOOL
if ! ID & ID
  ID -> ID = BOOL
```

(c) The abstraction that led to the false clone.

Figure 5: Examples that motivate our use of textual similarity as a filter.

stage (Section 4.3) extracts the syntactic differences between each clone as a set of fine-grained *change* operations, which we then classify as 'benign' or 'buggy,' the latter of which we present as bug reports.

4.1 Filtering

The most straightforward way of improving DejaVu's precision is through coarse-grained filtering of the inconsistent clones; that is, one can simply discard clones that do not meet certain criteria. In terms of recall, however, reckless and overly coarse filtering is antithetical to our goal of finding a large number of general clone bugs; overly selective filters are both difficult to generalize and often biased toward 'low hanging fruit.'

With this in mind, we limit DejaVu's coarse filtering to a single rule, a *minimum textual similarity* threshold that allows us to establish an important foundational assumption: that each clone pair we examine very likely originates from an *intentional* copy and paste operation. Our intuition is that inconsistent changes between incidentally similar code fragments are far less likely to be buggy than those between true deliberate clones. Concrete motivating examples for this filter appear in Figure 5.

Textual Similarity On manual inspection, a developer may be able to quickly dismiss false clones due to the (somewhat nebulously defined) intuition that the fragments do not 'look' similar at all. We codify and automate this process using a notion from the field of information retrieval: *document similarity*.

A similarity index of two documents is a value in the closed interval $[0,1]$ that describes their relative resemblance. A variety of approaches exist, but many follow a similar pattern: generate representative *sets* from each document and compare them using a set similarity measure. To compute textual similarity of source code, DejaVu generates for each code fragment a *w-shingling* [4], a set of contiguous, fixed-length subsequences of tokens the program's text, and compares the sets using the Jaccard index:

$$J(X,Y) = \frac{|X \cap Y|}{|X \cup Y|}$$

Example: Consider the following two lines of source code:

```
int res = NULL;
  int res = 1;
```

These lines correspond to the following shinglings of length three ('trigrams'):

$$F_1 = \{ \text{[int, res, =], [res, =, NULL], [=, NULL, ;] } \}$$
$$F_2 = \{ \text{[int, res, =], [res, =, 1], [=, 1, ;] } \}$$

Their intersection consists of the first element of each, and their union is of size five. The similarity of the two fragments is:

$$J(F_1, F_2) = \frac{1}{5} = 0.2$$

For our experiments, we use sequences of length five (5-grams), a length near the median number of tokens per line, and we set our threshold to the fairly permissive value of 0.5, which we arrived at after a series of exploratory experiments. There are more formal ways of fitting these thresholds: we can treat them as parameters and then use methods from statistics such as cross-validation and ROC curves to estimate the effectiveness of a setting and make a decision on the optimal values. For the purposes of our concrete dataset, the values above were sufficient.

Though we only implement a single filter, this stage is a natural extension point for project-specific tuning. For example, we noted in our experiments that a disproportionate number of false alarms came from a single large file: a remote procedure call client interface completely comprised of idiomatic marshaling and network code. Had the goal of our experiments been to produce as precise a report as possible rather than to evaluate our system, we could have implemented a filter to exclude this file and others like it.

4.2 Refinement

The next stage of our buggy change analysis is the *refinement* of individual clone pairs. The goal of this phase is twofold: we prune inconsistent, but likely spuriously-matching *prologues* and *epilogues* from each pair, and we simultaneously establish a common *abstraction* over which to compare the clones. We illustrate this process in Figure 6 and describe it here.

We first establish an abstraction for each code fragment using flexible lexical analysis, configured for the C++ lexicon (with preprocessor directives) with the following options:

- We fully abstract literals by type. All Boolean literals become a token marked '$BOOL,' for example.
- We perform consistent mapping of identifiers to normalized values. DejaVu greedily normalizes identifiers as they occur: the first referenced identifier of each fragment becomes '$0ID,' the second '$1ID,' and so on. For equal but consistently renamed code fragments, this generalization strategy produces a perfect substitution of identifier names. However, this strategy performs poorly in the presence of minor structural changes that introduce or reference *new* identifiers in otherwise equivalent fragments: it incorrectly causes a cascading chain of differences. We mitigate this effect by *not* normalizing the shared (*i.e.*, not renamed) identifiers and greedily renaming the remaining set.
- Whitespace and comment tokens are discarded, and keywords are left intact.

An example of this normalization appears in the lower row of Figure 6: note the lack of abstraction on the shared identifier `free`.

After abstraction, we *align* the two normalized sequences using a sequence differencing algorithm (Ratcliff/Obershelp [24]). If we find that a pair of inconsistent clones starts or ends with an inconsistent prologue or epilogue, respectively, we prune it: as described in Section 3, trivial extensions of clones are usually uninteresting. Although our earlier-described postprocessing solves this problem for the common case of trivial extensions to exact clones, we find that these spurious matches are quite common in highly divergent clones. Note that we are *not* throwing out the entire clones, we are merely refining—'trimming'—them to their essential core.

Figure 6 illustrates an additional subtlety: after pruning, the abstraction—namely, the identifier renaming—is no longer valid and must be repeated. We iterate this abstraction and alignment process until each clone is surrounded by a user-configurable amount of *exact context*, which we currently set to five tokens.

To evaluate our system's general applicability, we limited our configuration of the lexer to the three rules described above. However, additional project-specific configuration is possible. An embedded system project, for example, might selectively choose to abstract all bit-manipulating operators to a single value, while an enterprise system may prefer to not abstract string literals so as to identify embedded SQL bugs.

4.3 Buggy Change Isolation

At this step in the analysis, DejaVu has produced a set of inconsistent code fragments that are 1) very likely deliberately copied and 2) abstracted, aligned, and trimmed; that is, they are focused on their essential differences.

The final, and arguably most important, phase of our analysis takes these aligned, abstracted token sequences and distills them into a sequence of fine-grained *change operations*: insertions, deletions, and replacements that are capable of transforming one fragment in the clone pair into its inconsistent twin. DejaVu then classifies each of these individual changes as 'benign' or 'buggy.'

We perform this classification with a series of syntactic filters. Each filtering function has access to a variety of information: 1) the source text and abstracted tokens involved in the change, 2) the complete source text of both fragments, and 3) the set of *all* changes (for making contextual judgments). While we have implemented a small set of general default filters, we expect this natural extension point to be commonly utilized in practice: it provides a simple, intuitive, and fast point at which users can 'fit' DejaVu to a specific project and focus precisely on bugs of interest. Our three default filters are defined as follows:

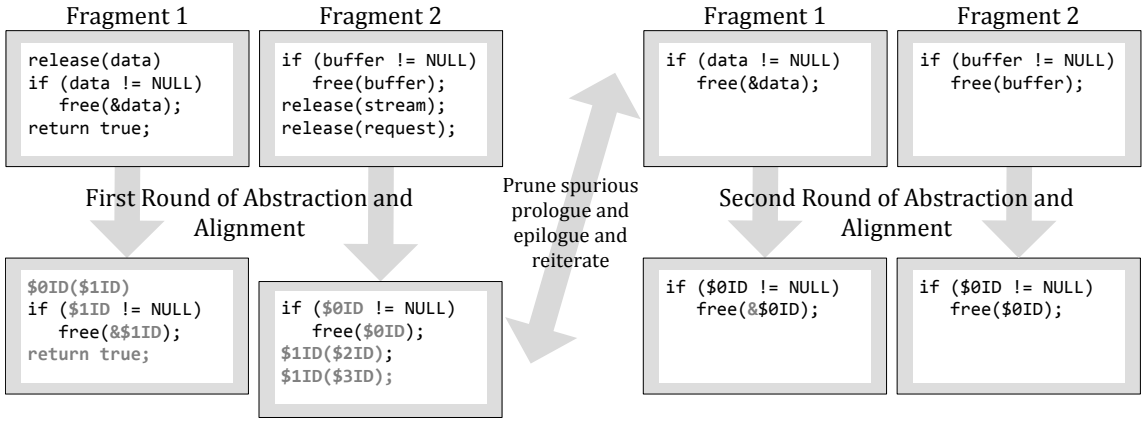


Figure 6: An example of our refinement process. Matching fragments are typeset in black and differences are highlighted in grey.

Punctuation Our first filter is simple: we discard changes that consist solely of punctuation (*e.g.* semicolons and braces), as they are usually not semantics-affecting.

Parameter Additions This filter controls for a specific class of adaptive change: the addition of new arguments to a function. An example of this filter's applicability appears in Figure 5a: on line 5 of each fragment, note that each calls a different function. This particular change is adaptive and a result of differing functionality, while the change at line 15 is the more relevant, suspicious change.

Identifier Scope This filter affects *only* changes that contain identifiers; all others implicitly pass. For any change that contains at least one identifier, at least one identifier must be *shared* with the pair clone, and at most one identifier can be *unshared*. Our intuition here is that divergent bug fixes are most likely applicable when their referenced variables are in scope. Note that we do not require all variables to be in scope: a common form of bug is that of a missing but necessary function call.

Our experiments were configured with exactly these three change filters. However, throughout our evaluation of the bug reports, we were without a shortage of ideas for additional filters, and our extensible framework simplifies the process of implementing and evaluating them.

Reporting Once the buggy changes have been identified, DejaVu outputs each clone pair containing at least one suspicious change as a bug report. Along with the source text, the report entries include a rendered diff of the *specific* changes DejaVu finds suspicious: DejaVu maps each change back to the relevant source lines, displaying 1) any line in its entirety that contains at least one structural difference and 2) between lines, precise intra-line markers highlighting the locations of relevant structural differences. Note that in our evaluation, we only marked a bug as 'true' if DejaVu identified both the relevant clone pair and the relevant buggy change(s).

These differences highlight the essence of the inconsistent edit and facilitate inspection: in our evaluation (*cf.* Section 5), each bug report took an average of 2.4 minutes to verify, albeit with a somewhat high variance. Note that DejaVu often finds bugs that have already been *fixed* at least once, so the solution to the problem is often encoded in the bug report.

Overfitting and Generality A natural objection to the specific instances of the techniques described above is *overfitting*: we may have developed filters and rules that are specific to our current target. While the particular rules we present are arguably general, we do not see generality as an absolute requirement: our framework has several extension points, and project-specific filters are simple to write and try. The development of a purely syntactic bug finding analysis that functions equally well on all projects is likely a futile exercise: in its most practical setting, we envision DejaVu's use in a workflow in which aggressive, project-specific filters are used and refined frequently.

Scalability These operations are potentially expensive to compute: each involves numerous string and set operations that are potentially quadratic in the size of the code fragments. However, in practice, the buggy change analysis module executes quickly. The key factor influencing its scalability is the massive reduction in problem size afforded by the scalable clone analysis. In our experiments, the clone detection front end enumerates $O(10^8)$ source code fragments. Pairwise analysis of these fragments for buggy changes would require $O(10^{16})$ expensive executions of the buggy change analysis framework. Instead, the clone analysis—despite being tuned for recall—reduces this to a much more manageable set of $O(10^4)$ clone pairs, each requiring only a single expensive change analysis.

Nonetheless, this module has numerous opportunities for optimization. First, it is implemented in Python and would achieve a significant gain by moving to a more performance-oriented language. Second, this task is easily parallelized, as it involves only local comparisons between two code fragments. Third, more scalable but less precise similarity techniques like winnowing [26] may provide a further increase in performance by quickly removing the least similar clones.

Parameter	Value
Minimum code fragment size	6 statements
DECKARD 'similarity' value	0.8
Minimum textual similarity	0.5
Textual similarity n-gram length	5 tokens
Alignment: min. exact context	5 tokens

Table 1: DejaVu's parameter settings.

DejaVu Execution Time (Elapsed)		
Component		Time (hh:mm)
Clone Detection	Sequential Portion	0:45
	Parallel Portion	2:05
Buggy Change Analysis Framework		1:30
	Total	4:20

Table 2: Execution time on our 75+ million line code base.

5. Experimental Results

An execution of DejaVu on our large commercial code base produced 8,103 bug reports in under five hours. We estimate that 2,070–2,760 of these very likely represent true, previously-unknown bugs. The following sections provide a detailed presentation of this evaluation as well as a quantitative and qualitative assessment of the results.

5.1 Methodology and Timing

We evaluated DejaVu on a code base with 100+ million lines of C and C++ source code.[3] 75+ million lines are contained in actively built source files and 25+ million lines reside in header files. This code base comprises approximately 450 projects that form a single larger product family. Despite this unifying relationship, the individual projects cover a diverse range of functionality.

The front end of our tool resides within the compiler and thus considers only actively compiled code. This is an interesting property of DejaVu, which can be considered both a strength and a weakness. On the positive side, errors in uncompiled code are likely to be of low severity and may be considered noise. However, should we need to process files outside of a functional build system, the compiler that hosts our front end does have the ability to run its parsing stage in isolation (*i.e.*, "syntax check only" mode), which does not check dependencies and can be run with a minimum of effort. In practice, we intend to use DejaVu only on actively compiled code as part of the regular development process.

Timing We deployed DejaVu on a single master node and four cluster nodes. The master node is equipped with a dual core Intel processor and Windows Server 2008. Each cluster node is equipped with two dual core AMD Opteron processors and a 64 bit version of Windows Server 2003. The experimental parameters used for DejaVu's various components are described throughout this paper; we consolidate them for reference in Table 1.

We ran DejaVu's front end separately, during a normal build of the code base. The integration with the custom build system rendered precise overhead measurements difficult, but we observed that the generation of clone candidates added a negligible amount of time to the overall build. In addition, future executions of this step, which are executed incrementally through a build system, isolate this slight amount of overhead to changed files only.

The rest of the process, which includes clone detection, change analysis, and report generation, took a total of 4 hours and 20 minutes. A breakdown of the time spent in individual components appears in Table 2. The clone detection phase was configured with a fairly liberal similarity setting: the mean edit distance between the clones in our result set is 28 tokens, and controlled for size, the clone pairs differ by approximately 14% on average (with a fairly high variance). Despite this potential scalability trap, the clone detection phase completes in under 3 hours and is dominated by the parallel step, which we execute on all 16 available cores (4 nodes $\times$ 2 CPUs/node $\times$ 2 cores/CPU). Adding more nodes to the cluster is likely to improve performance further, and as discussed in Section 4, the final refinement step contains numerous opportunities for optimization.

5.2 Error Reports

DejaVu produced 8,103 bug reports, which were beyond our resources to verify fully. We instead sampled 500 reports uniformly at random—a significant sample size—and divided the set into five categories:

Bugs Very likely bugs. Examples are found throughout this paper.

Code Smells Suspicious but ultimately harmless differences that are often inconsistently fixed. Example: a dead store to a local variable that one programmer notices is redundant and deletes from his or her copy.

Style Smells Changes in style that leave the semantics of the code fragments unaffected but may hurt clarity. For example, some code fragments contain clever uses of assignments as expressions. When copied, a programmer may adapt the fragment for clarity using temporary variables.

Unknown There is nothing apparently false about these reports, and the necessary conditions for their validity are there; the changes apparently apply. However, we could not confidently judge them as bugs without further domain knowledge.

False False reports. These consist mainly of adaptive changes we fail to account for and filter, but also occasion-

[3] We have also performed an additional experiment on an open source code base; it is included as Appendix A.

Category	Count	Rate	Extrapolated Count
Bugs	149	30%	2,070 – 2,760
Code Smells	49	10%	450 – 1,140
Style Smells	60	12%	630 – 1,320
Unknown	58	12%	600 – 1,280
False Reports	184	37%	2,640 – 3,330

Table 3: Categorization of sampled bug reports and estimates of total counts with 95% confidence.

ally include pathological cases in which every inconsistent edit is intentional.

The results of this evaluation appear in Table 3. Our sample size of 500 allows us to estimate the rate of any one of these parameters in our larger report with a $\pm 4.25\%$ margin of error with 95% confidence (equivalently, a $\pm 3.56\%$ margin with 90% confidence). Conservatively assuming every 'Unknown' report is false, our overall 'true-bug' precision is very likely between 26 and 34%: *triple* that of previous work, and our set of bug reports very likely contains 2,070–2,760 true bugs. Weakening our definition of 'bug' to include code smells—but still conservatively assuming all 'Unknown' reports to be false—raises our precision to between 48 and 56%.

Qualitative Assessment The potential bugs are quite varied: Figures 7 and 8 contain two of the more interesting examples. In the first example (Figure 7), locking around accesses to a field are added to otherwise identical functions. In the second example (Figure 8), a file resource is leaked along an error path. The bugs presented earlier in this paper (Figures 1 and 5a) are equally diverse. These examples highlight particularly well DejaVu's novelty: we are able to detect a very general class of clone bugs.

Many of the bugs are more mundane. Figure 9 contains three common classes of bugs that we observed repeatedly in the result set. The first (Figure 9a) is a general error related to input validation: the author mistakenly joined the two disjoint error conditions with the logical 'and' operator. This example is particularly obvious; line 2 of the first fragment simplifies to `false`. The second example (Figure 9b) involves a common idiom: function error codes are commonly collected in a single 'result' variable and returned upon exit. We observed many cases where this was omitted—likely by mistake—and fixed inconsistently. The final example (Figure 9c) is the ubiquitous null check: we found numerous examples where inconsistent null checks were added to otherwise textually identical code fragments—even within a single file.

Code Smells and *Style Smells* also comprised a significant proportion of the results and are worth exploring in that they can often lead to *future* bugs. Three common examples appear in in Figure 10. Other examples include unnecessary declarations, inconsistent type qualifiers (*e.g.*. `const` and `unsigned`), and unnecessary explicit conversions from non-Boolean types to Boolean values.

False Positives At present, about 50% of the potential bug reports are either obviously false or unverifiable by us without more extensive domain knowledge. As we noted in Section 4, many of these could be filtered at the risk—or perhaps the benefit—of overfitting DejaVu to our current target code base. This number is quite manageable, and unlike in many forms of traditional static analysis, these 'false' reports are beneficial to investigate: duplicate code is a 'code smell' in its own right. Our analysis framework ensures that all the false positive *bug* reports are still almost certainly derived from true *clone* reports, so the false positives are often viable refactoring candidates.

Lastly, as reports are confirmed by the development teams, we expect to implement several filters that provide more *relevant* results; that is, we intend to tune DejaVu to not only find true bugs, but to also focus on the bugs the developers believe are severe. Such filters would act at the expense of recall; we present only the most general results in this paper.

6. Discussion and Related Work

In this section, we provide a detailed comparison with other copy-paste bug detection tools as well as a brief overview of other related work.

6.1 Copy-Paste Bug Detection Tools

Despite a vast amount of research on clone detection, comparatively few tools actively detect clone-related bugs; Li *et al.*'s CP-Miner [21] and Jiang *et al.*'s context-based bug detection tool [16] are the best-known examples.

CP-Miner CP-Miner [21] detects *identifier renaming* bugs. Identifier renaming bugs occur when a developer copies a code fragment and intends to rename one or more identifiers consistently but omits a case. Though these bugs are often caught by the compiler, they are quite difficult to trace when they do occur. Our tool succinctly captures this class of bugs as well: we start by fully abstracting identifier names, finding both consistently and inconsistently renamed identifiers, and later detect inconsistent renaming problems as structural differences in the token streams. The method by which we unify variable names is quite similar to that used by Li *et al.*, though their more specific heuristics are likely much more precise for this class of bugs.

For comparison, we conducted an additional simple study: we excluded from our report clones who differ only in the consistent naming of identifiers. We noted only an approximate 4% decrease in our report size, which suggests that our general technique is finding a significantly larger new class of bugs. Within that 4%, we noted many of the same types of false positives reported by Li *et al.*, including simple reorderings. As part of our ongoing work, we intend to integrate their heuristics for this class of bugs into our own tool.

Context-Based Jiang *et al.* developed a tool [16] that looks for exact clones that are *used* inconsistently. The intuition is that duplicate code should be used in similar contexts, *e.g.*

```
CS.Enter();
request = m_request;
m_request = 0;
CS.Leave();

if (request) {
  CloseRequest(request);
  request = 0;
}
/* Body of method omitted for space (20 lines) */
CS.Enter();
if (IsTaskCancelled())
  Log(USER_CANCEL);

m_request = request;
request = 0;
CS.Leave();

Error:
CS.Leave(); // Force leave here if still inside CS.

CloseRequest(request);
return res;
```

```
request = m_request;
m_request = 0;


if (request) {
  CloseRequest(request);
  request = 0;
}
/* Body of method omitted for space (20 lines) */

if (Cancelled())
  Log(USER_CANCEL);

m_request = request;
request = 0;


Error:

CloseRequest(request);
return res;
```

Figure 7: A concurrency bug. Though in a different class, the right fragment contains the critical section field *and* guards other accesses of `m_request` with it.

```
FILE file = Open(fName, WRITE, CREATE, NORMAL);
if(file == INVALID_FILE) return E_FAIL;

int size = SizeofResource(mod, rsc);
int written;
RESOURCE rGlobal = LoadResource(mod, rsc);

if(!rGlobal)
   return E_FAIL;

void *data = AccResource(rGlobal);
if (data == NULL) {
   res = E_FAIL;
   goto exit;
}
if (!WriteFile(file, data, size, &written, NULL)) {
   res = E_FAIL;
   goto exit;
}
exit:
if(file != INVALID_FILE_VALUE)
   Close(file);
```

```
FILE file = Open(fileName, WRITE, CREATE, NORMAL);
if(file == INVALID_FILE) return E_FAIL;

int size = SizeofResource(mod, rsc);
int written;
RESOURCE rsrc = LoadResource(mod, rsc);
if(rsrc == NULL) {
   res = E_FAIL;
   goto Cleanup;
}
void *data = AccResource(rsrc);
if (data == NULL) {
   res = E_FAIL;
   goto Cleanup;
}
if (!WriteFile(file, data, size, &written, NULL)) {
   res = E_FAIL;
   goto Cleanup;
}
Cleanup:
if(file != INVALID_FILE_VALUE)
   Close(file);
```

Figure 8: A resource leak. The file is not closed on all paths.

a null check surrounding one clone should also surround another. The authors report about a 10% true positive rate overall. Though we find many of the same kinds of bugs—indeed, local instances of context inconsistencies are a special case of our general inconsistencies—the tool is complementary to our own in general, as the surrounding context of a clone can often be quite distant and thus undetectable as a single inexact clone. Like this tool, DejaVu operates over syntax trees, so we can integrate this technique with our own by recording the enclosing syntactic context of each clone and using it as a clone-related feature for filtering.

6.2 Other Related Work

Clone Detection Tools Many tools have been developed for detecting similar code fragments, each with varying levels of scalability, abstraction, and granularity; we briefly present a sampling of the more scalable techniques for comparison.

Baker's tool 'dup' [2] was one of the first clone detection tools and it remains one of the most scalable. It operates at line granularity and allows for consistent renaming of tokens via *parameterized string matching*. It is incapable, however, of finding clones with structural differences.

```
// Check 'level' function argument
if( level < 1 && level > NUM_LEVELS )
   return INVALID_ARG;
```

```
// Check 'level' function argument
if( level < 1 || level > NUM_LEVELS )
   return INVALID_ARG;
```

(a) Poor argument checking. NUM_LEVELS is a positive integer constant.

```
// Finally, submit the message !
SubmitMessage(msgStatus);
/* ... */
return result;
```

```
// Finally, submit the message !
result = SubmitMessage(msgRoute);
/* ... */
return result;
```

(b) Not saving the result of an important function call.

```
if (!AddItem(psp, pspDest, *desc, AFTER))
    return false;
```

```
if (!desc || !AddItem(psp, pspDest, *desc, AFTER))
    return false;
```

(c) An added null check to an otherwise identical pair of larger functions.

Figure 9: Three examples of common classes of simple bugs. Context is omitted for brevity.

As discussed throughout this paper, Deckard [15] forms the basis of our approach. Gabel *et al.* later extended this technique [11] to scalably find clones within subgraphs of *program dependence graphs* [10], which encode data and control flow information. The dataflow information encoded in PDGs may improve the precision of DejaVu's bug reports by filtering non-semantics-affecting changes in a more principled way.

Li's CP-Miner [21], discussed earlier as a bug finding tool, is also based on a novel clone detection algorithm. Like DejaVu, CP-Miner enumerates a set of possible clone candidates and later groups them together. CP-Miner first enumerates and abstracts the *basic blocks* of the target code base and then uses a data mining technique, *frequent subsequence mining*, to enumerate the possible clones. This approach allows for the detection of clones with an inserted or removed statement (referred to as a *gap*). To output maximal clones, CP-Miner assembles the smaller basic block-level clones into larger, more complete copied code segments.

CCFinder [18] uses a suffix tree-based algorithm to find clones within token streams. Though asymptotically quadratic, it scales well to software projects of a few million lines of code, though it is incapable finding of any form of structurally inconsistent clone. A later parallelized extension, D-CCFinder [22], scales this approach to a 400 million line code base, which takes 2 days to complete on 80 nodes. The

```
res = chunk->put_Context(chunkContext);
ON_ERR_RETURN(res);
res = chunk->put_Locale(chunkId);
ON_ERR_RETURN(res);
```

```
res = ((Chunk*) chunk)->put_Context(chunkContext);
ON_ERR_RETURN(res);
res = chunk->put_Locale(chunkId);
ON_ERR_RETURN(res);
```

(a) A strange extraneous cast.

```
PSESSION pSession;
if (NULL == (pSession = GetActiveSession()))
  return;
```

```
PSESSION pSession = NULL;
if (NULL == (pSession = GetActiveSession()))
  return;
```

(b) A dead store to a local variable.

```
len = StringLen(bstr);
if (wz == NULL || *outlen < (len + 1))
   {
   *outlen = StringLen(bstr);
   return wz == NULL;
   }
```

```
len = StringLen(bstr);
if (wz == NULL || *outlen < (len + 1))
   {
   *outlen = len;
   return wz == NULL;
   }
```

(c) Redundant recomputation of a value.

Figure 10: Three code smells.

extension is straightforward: they divide the problem into n smaller chunks and find clones within the $\binom{n}{2}$ possible pairings. Though on the same level of scale as DejaVu, it does not find inconsistent clones or attempt to find clone bugs.

Schleimer *et al.*'s MOSS [26] is a highly scalable system for detecting software plagiarism. It utilizes a document similarity technique related to the algorithms we apply in our inspection framework. However, it is quite coarse grained, operating at the program or file level, and is thus unsuitable for finding clone related bugs.

Studies of Cloning Each of the above tools reports that a significant amount (5-30%) of their target systems consist of copied code, which motivates well our technique. Kim *et al.* [20] study the nature of code clone *genealogies* by utilizing a project's version control history to track the introduction, evolution, and removal of clones over time. In a portion of the work that is relevant to our own, they find that a significant proportion of all code clones over the history of two software projects evolved in sync and are thus prone to the general class of errors that we detect.

In a concurrently developed study [17], Juergens *et al.* extend a suffix tree-based clone detection algorithm to detect

clone pairs with a small number of edits. They report the inconsistent clones in their entirety to the developers of several proprietary software projects and study the fault rate of the inconsistent clone fragments. They find that these fragments are more failure-prone than non-copied code, which validates our own approach. However, their algorithm is asymptotically quadratic and appears to hit its scaling limit around 5 million lines of code.

Lightweight Bug Finding Engler *et al.* [8] introduced the idea of using behavioral inconsistencies to find bugs. For example, if a pointer is dereferenced unconditionally on one program path but is checked for validity on another, their tool outputs a warning: either the check is unnecessary or a bug exists. This is quite similar in spirit to our own work: we search for bugs using *syntactic* inconsistencies. Like Engler *et al.*'s work, our work inherently exposes the nature of the inconsistencies, greatly simplifying the repair of the defects.

Other tools, like the historical LCLint [9], FindBugs for Java code [14], and Engler *et al.*'s Metal specification language and checker for systems code [7, 13] are all capable of using syntactic characteristics and/or a local static analysis to find bugs in large software systems. (Interestingly, in a later empirical study [5], Chou *et al.* note that a large proportion of the errors found by their tool are propagated by copying and pasting source code.) These tools are all alike in that they require as input explicit specifications for specific classes of bugs.

In our scenario, we expect copied code to form a type of meta-level implicit specification: duplicate code is generally intended to remain identical. We extract violations of this 'implicit specification' by leveraging an efficient, large scale clone analysis, eliminating the need to specify and check instances of duplication manually. Though this higher level specification is weaker, admitting many false positives, it captures a much more general category of bugs.

7. Conclusions and Future Work

In this paper, we have presented DejaVu, a system for locating general copy-paste bugs. We demonstrated that DejaVu is highly scalable: a full analysis completes in under five hours on 75+ million lines of source code. Our system located 2,070–2,760 very likely bugs and we have shown that these bugs are general and varied, affecting resource leakage, concurrency, memory safety, and other important classes of software errors.

Our most immediate future work involves continued evaluation of our tool, the first aspect of which is continuing our communication with the developers of our target system. We are currently gathering confirmation of bug reports and soliciting feedback on the tool's output. We also intend to run DejaVu against several other large code bases, both commercial and open source.

References

[1] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Proceedings of FOCS '06*, 2006.

[2] B. S. Baker. On finding duplication and near-duplication in large software systems. In *WCRE '95: Proceedings of the Second Working Conference on Reverse Engineering*, 1995.

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

[4] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. In *Selected papers from the sixth international conference on World Wide Web*, pages 1157–1166, 1997.

[5] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, 2001.

[6] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, 2007.

[7] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*, 2000.

[8] D. R. Engler, D. Y. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *SOSP*, 2001.

[9] D. Evans, J. Guttag, J. Horning, and Y. M. Tan. Lclint: a tool for using specifications to check code. In *SIGSOFT FSE*, 1994.

[10] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3), 1987.

[11] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, 2008.

[12] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proc. VLDB*, 1999.

[13] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proc. PLDI '02*, 2002.

[14] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *OOPSLA '04*, 2004.

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

[16] L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. In *ESEC-FSE '07*, 2007.

[17] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *ICSE '09: Proceedings of the 31st international conference on Software engineering*, 2009.

[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), 2002.

[19] C. Kapser and M. W. Godfrey. "Cloning considered harmful" considered harmful. In *Proc. WCRE '06*, pages 19–28, Washington, DC, USA, 2006. IEEE Computer Society.

[20] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *ESEC/FSE-13*, 2005.

[21] 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 *OSDI*, 2004.

[22] S. Livieri, Y. Higo, M. Matushita, and K. Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder. In *ICSE '07*, 2007.

[23] T. T. Nguyen, H. Nguyen, N. Pham, J. Al-Kofahi, and T. Nguyen. Cleman: Comprehensive clone group evolution management. *Automated Software Engineering (ASE), 2008.*, Sept. 2008.

[24] J. W. Ratcliff and D. Metzener. Pattern matching: The gestalt approach. *Dr. Dobb's Journal*, July 1988.

[25] C. K. Roy and J. R. Cordy. An empirical study of function clones in open source software. In *WCRE '08: Proceedings of the 2008 15th Working Conference on Reverse Engineering*, 2008.

[26] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *SIGMOD*, 2003.

[27] M. Toomim, A. Begel, and S. Graham. Managing duplicated code with linked editing. In *Proc. IEEE Symp. Visual Languages: Human Centric Computing*, 2004.

[28] R. Yang, P. Kalnis, and A. K. H. Tung. Similarity evaluation on tree-structured data. In *SIGMOD*, 2005.

A. Appendix: Experiment on Mozilla

To further demonstrate DejaVu's general effectiveness as a bug finding tool, we performed an additional experiment on an open source code base owned by the Mozilla Foundation. This appendix summarizes the results.

A.1 Setup

Code Base We gathered current Mercurial (a distributed version control system) snapshots of the source code for the Mozilla Firefox web browser (`http://mozilla.com/firefox`), the Thunderbird email client (`http://mozilla.com/thunderbird`), and their supporting libraries. Using a naïve methodology, we counted approximately two million lines of non-header C and C++ source code.

Building After gathering the source code, we integrated DejaVu's front end into the Mozilla build process and ran two standard builds, one each for Firefox and Thunderbird. In this configuration, DejaVu only scanned code actively compiled during a standard Windows platform build of each project.

Configuration We used the same experimental parameters as those described in our formal evaluation (Section 5), with one notable exception: due to the much smaller size of this code base, we performed the entirety of the experiment on a single node, a 2.4 GHz Core 2 Duo system with 4 GB of RAM running Microsoft Windows 7. We configured DejaVu's clone analysis to use both available cores.

A.2 Results

Timing Running on a single, node, DejaVu's clone detection back end completed in 40 minutes and the buggy change analysis framework completed completed in four minutes. For a code base of this size, scalability is not an issue.

Results DejaVu produced 136 bug reports. We categorized each report using the same methodology and classification scheme as in our formal evaluation. (Any statistically significant random sample of a population this small would include nearly every report; we thus opted to verify them all.) The results appear in the following table.

Category	Count	Rate
Bugs	45	33%
Code Smells	21	15%
Style Smells	11	8%
Unknown	33	24%
False	26	19%

Table 4: Categorization of Mozilla bug reports.

We note a comparable rate of true bugs and similar rates for the other categories: DejaVu appears to be generally effective on this code base as well. However, we tended to annotate bugs as 'Unknown' at nearly twice the rate as in our commercial code base. This is likely due to our lack of previous exposure to the Mozilla source code.

The complete experimental results, including DejaVu's output and our classifications, can be found online at the following address: `http://wwwcsif.cs.ucdavis.edu/~gabel/research/dejavu_mozilla.zip`