

# Tracking Your Changes: A Language-Independent Approach

Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta, *University of Sannio*

Based on a novel differencing algorithm, this approach to tracking source code evolution in real-world software systems achieves acceptable precision and overcomes the Unix diff's versioning limitations.

**V**ersioning and bug-tracking systems are invaluable assets for large software projects that involve developers spread worldwide and numerous users reporting bugs and proposing enhancements. In addition to supporting development, versioning systems are a precious source of information for studying or monitoring a software system's evolution.

Such monitoring requires tracking source code artifacts across file revisions. Versioning systems perform this tracking using the Unix diff algorithm, which, to preserve space, treats any change as the minimum sequence of additions and deletions. This process limits diff's ability to distinguish line additions and removals from line changes. If a source code line changes, the versioning system records it as the removal of the old line and the addition of the new one. This makes it difficult to discern whether the change only partially modified the old line or completely replaced it.

A degree of subjectivity is inherent in such a classification because humans can disagree on the extent to which something has changed. Still, we foresee differencing algorithms that can make these classifications within a reasonable margin of error.

We present an approach for tracking software entities across multiple revisions of a file that relies on our recently introduced language-independent differencing algorithm.<sup>1</sup> This algorithm overcomes diff's limitations and, unlike algorithms based on the code Abstract Syntax Tree (AST),<sup>2</sup> doesn't require the code to be parsed. This makes the algorithm suitable, if a detailed classification of the change isn't needed, for tracking the evolution of software entities treatable as a sequence of lines, such as source code, but also requirements, use cases, and test cases.

We implemented the differencing algorithm in a tool named *ldiff* (*line differencing*), which has syntax and output similar to the Unix diff. This compatibility means that the tool's use by diff users and integration in integrated development environments (IDEs) or recommender systems relying on a Unix-diff-like algorithm can be immediate. (For a detailed description of *ldiff*, see the "Ldiff: A Support Tool" sidebar, and, for additional approaches to tracking, see the "Related Work in Differencing Algorithms" sidebar.)

## Software-Entity Tracking

A software entity is a set of (not necessarily adjacent) lines from a textual artifact, such as a source code clone, a source code line containing a vulnerable statement, or a comment. Tracking the evolution of an entity within a file comprises four stages that aim to identify the units of analysis, identify the entities to be tracked, track the entities, and identify entity changes.

### Identify the Units of Analysis

A software entity's evolution can be apparent across system releases or single file commits, or by grouping logically related commits. As Harald Gall and his colleagues proposed, a software system's evolution is viewable as a sequence of snapshots.<sup>3</sup> These snapshots, they contend, develop from *change sets*,

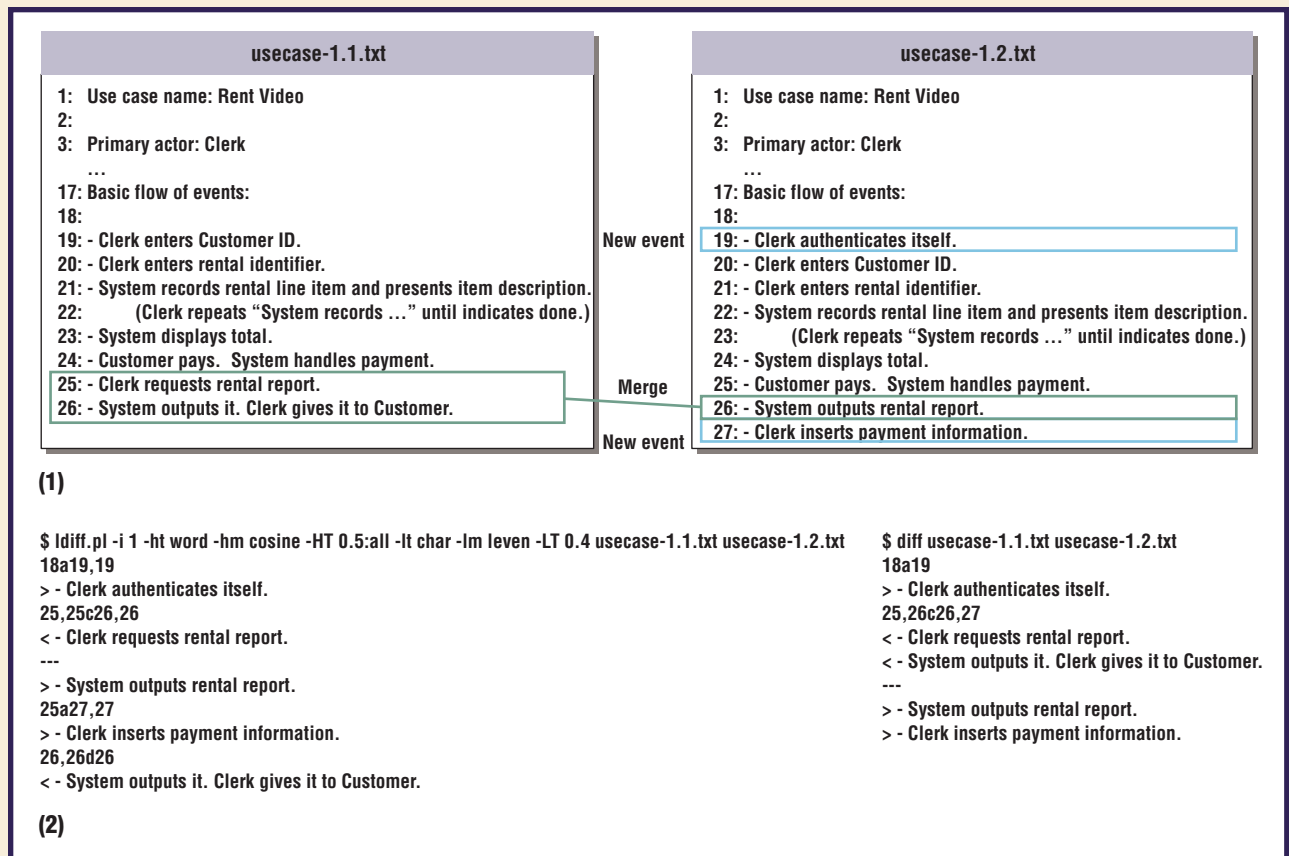
## Ldiff: A Support Tool

Our approach to tracking software entities relies on ldiff, a differencing algorithm that has syntax and output similar to the Unix diff. A Perl implementation of our differencing algorithm is available at <http://rcost.unisannio.it/cerulo/tools.html>. The tool supports various hunk similarity metrics (cosine, Jaccard, dice, and overlap) and text item extraction techniques (chars, words, n-grams, and C/C++/Java language tokens). Figure A1 shows an example of using both ldiff and diff on two versions of a text artifact (use case).

In Figure A2, we invoked ldiff with one iteration (-i 1), a

word hunk tokenizer (-ht word), a cosine hunk similarity measure (-hm cosine), a hunk similarity cut-level threshold of 0.5 (-HT 0.5:all), a char line tokenizer (-lt char), a Levenstein line similarity measure (-lm leven), and a line similarity threshold of 0.4 (-LT 0.4). We invoked the Unix diff command with its default parameters.

The Unix diff was unable to detect the last added use-case event, as it maps different adjacent lines into a single block change. Ldiff instead considered the added event as an addition, and the merged events as a deletion, combined with a change.



**Figure A. Comparing approaches. (1) We used ldiff and diff on two versions of a text artifact. (2) The Unix diff was unable to detect the last added use-case event.**

which represent the logical changes a developer performs in terms of added, deleted, and changed source code lines. Various approaches are available for reconstructing such a sequence. We chose the method that Thomas Zimmermann and his colleagues developed.<sup>4</sup> Their approach considers the sequences of file revisions that share the same author, branch, commit notes, and so on, such that the difference between the time stamps of two subsequent commits is 200 seconds or less.

### Identify the Entity of Interest

Before tracking an entity's evolution, we must first identify the entity of interest in the source code or textual file. The process can be manual (for example, the developer identifies a source code entity to track) or automatic (as with clones or vulnerabilities, which we'll discuss in a moment). Either option identifies the entity as a set of textual or source code file lines. The next stages will track the changes in these lines.

## Related Work in Differencing Algorithms

Miryung Kim and David Notkin classify code-differencing algorithms into algorithms working on a structured representation of the program, such as an Abstract Syntax Tree (AST), and algorithms working on a flat representation, such as a sequence of lines.<sup>1</sup>

Beat Fluri and his colleagues' work, Change Distiller, belongs to the first class and is able to detect with high precision the nature of structural changes that have occurred in the source code.<sup>2</sup> Our line-differencing algorithm can't identify these changes. Indeed, the value of change information obtainable from AST-based algorithms, such as identifying method signature changes and changes in class hierarchies, is high. However, the inherent computational complexity might limit their application in large-scale systems that have undergone numerous changes. In addition, a parser must be available, and system snapshots must be analyzable with that parser. These conditions might or might not be satisfied, because sometimes developers can leave the system in an incomplete or inconsistent state, such as with missing files or wrong links.

Steven Reiss showed that source code is trackable through multiple versions of a file by using relatively simple techniques, such as line matching based on the Levenstein distance.<sup>3</sup>

Zhenchang Xing and Eleni Stroulia introduced UML Diff, a differencing tool that can capture differences between UML models, identifying the addition, removal, or change of elements such as methods, attributes, and packages.<sup>4</sup> UML Diff is more specific than ldiff and, like Change Distiller, is more suited for performing detailed change analyses.<sup>2</sup>

Michael Godfrey and Lijie Zou proposed a method for detecting merging and splitting of source code entities that considers entities' various metrics and callee/caller relationships between entities.<sup>5</sup> Their approach is more specific than ldiff for studying refactoring activities, such as merging and splitting. ldiff focuses on tracking sets of lines across file revisions and, above all, on distinguishing line changes from additions and deletions.

### References

1. M. Kim and D. Notkin, "Program Element Matching for Multi-Version Program Analyses," *Proc. 2006 Int'l Workshop Mining Software Repositories (MSR 06)*, ACM Press, 2006, pp. 58–64.
2. B. Fluri et al., "Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction," *IEEE Trans. Software Eng.*, vol. 33, no. 11, 2007, pp. 725–743.
3. S.P. Reiss, "Tracking Source Locations," *Proc. 30th Int'l Conf. Software Eng. (ICSE 08)*, IEEE CS Press, 2008, pp. 11–20.
4. Z. Xing and E. Stroulia, "Differencing Logical UML Models," *Automated Software Eng.*, vol. 14, no. 2, 2007, pp. 215–259.
5. M.W. Godfrey and L. Zou, "Using Origin Analysis to Detect Merging and Splitting of Source Code Entities," *IEEE Trans. Software Eng.*, vol. 31, no. 2, 2005, pp. 166–181.

### Track Source Code Entity Changes

At the core of our proposed tracking approach is a line-differencing algorithm (LDA). An LDA treats source code as an ordered sequence of text lines and computes the differences between two revisions of a source code entity without considering the underlying syntax. It can provide information at the line level only as follows:

- the line was unchanged between revisions  $r_i$  and  $r_i + 1$ ,

- the line was changed between  $r_i$  and  $r_i + 1$ ,
- the line was present in  $r_i$  and then deleted in  $r_i + 1$ , or
- the line was added in  $r_i + 1$ .

The Unix diff is one of the most widely known LDA algorithms. It treats lines as a sequence of deleted and added lines, but it can't detect changed lines. When diff finds a sequence of additions and deletions, starting from the same position in the file, it assumes that the block has changed. However, this might not be the case. A completely different block could have replaced the original block, or part of it. If we consider, for example, source code lines 2–4 in the top left of Figure 1, changing as shown in the top right, the Unix diff would produce the following output:

```
2,4c2,3
< int b[];
< foo(c,b);
< if (size(b)>0) printf("D");
---
> int b[]={1,2};
> b=foo(c,b);
```

This output indicates that the block comprising three lines has changed into a block of two lines. However, it's highly unlikely that this was the case. Instead, it's very likely that the programmer has removed the third line and changed the first and second lines of the first block into the first and second of the second block. No automatic differencing tool would be able to unambiguously distinguish changes from the programmer's addition and removal, but a tool that can suggest likely changes would be desirable.

To this end, we proposed an LDA particularly suitable for code tracking.<sup>1</sup> Consider again the top-left source code fragment in Figure 1, evolving as shown on the right side. The algorithm starts (step 1) by applying the Unix diff to identify unchanged lines. Of course, the question arises of whether the unchanged lines that diff detected are a good approximation of the set of actually unchanged lines. The Unix diff searches for the longest of the many possible common subsequences between the two file releases. This reflects the assumption that a programmer tends to minimize the change effort by reusing existing lines, although this might not be the programmer's real intention.

Step 2 compares the *hunks*—that is, sequences of adjacent lines—of L and R that aren't classified as unchanged (highlighted in purple and tan, respectively). Specifically, the algorithm uses a hunk

similarity measure to compare all possible pairs of L and R hunks. Then, in Step 3, the algorithm takes the topmost *HT* (hunk threshold) distinct hunk pairs and, for each pair, performs a line-by-line comparison. In other words, the algorithm uses a line similarity measure to compare lines of the left-side hunk with those of the right-side hunk. The algorithm classifies line pairs with a similarity above a given line threshold *LT* as changed lines (green lines in Figure 1). All sets of adjacent lines not classified as changed are new hunks for a subsequent iteration of Steps 2 and 3.

More iterations increase recall, which we'll discuss in a moment, because subsequent iterations will consider combinations of hunks previously discarded. This process, for example, would be useful for detecting merge and split, in which each iteration will match only one source (in the case of merge) or one target (in the case of split). As the "Ldiff: A Support Tool" sidebar shows, *HT*, *LT*, and *i*, the number of iterations, are parameters a user can specify to calibrate the tool.

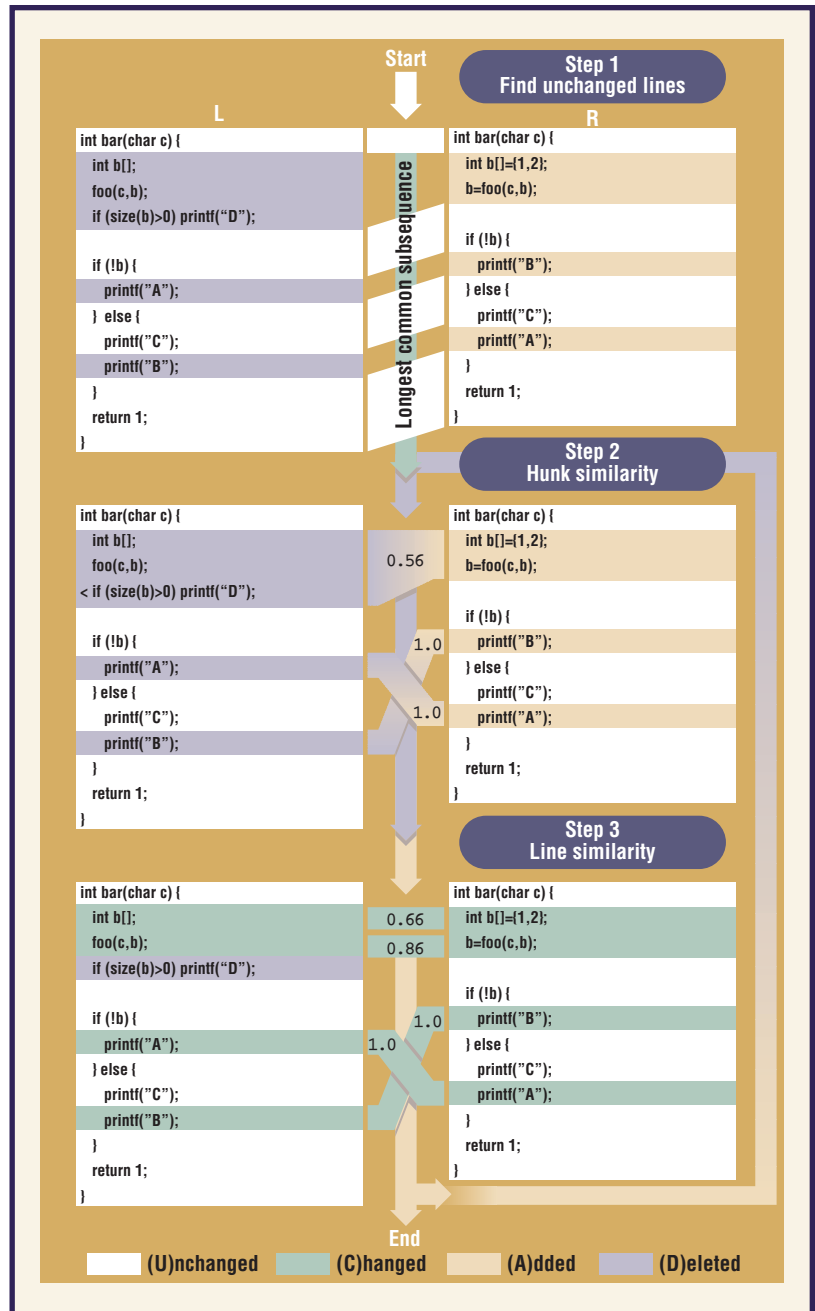
Hunk and line similarity metrics work on a finite set of items extracted from the text, such as characters, words, or tokens. *Set-based metrics* don't consider ordering information and are suited to compute hunk similarity. *Sequence-based metrics* take into account the order in which items appear and are suited for line similarity. Table 1 (on the next page) shows examples of metrics for computing these similarities. Set-based metrics compute hunk similarity (Step 2 of Figure 1), while sequence-based metrics (also known as distances) compute line similarity (Step 3 of Figure 1). In the studies and examples we discuss in this article, we always use the cosine set-based metric on words and the Levenstein sequence-based metric on characters.

### Identify Changes in Software Entities

By relying on the information from the previous steps, we can determine (see Figure 2 on the next page) whether

- a new software entity of interest appears in a given snapshot,
- an entity's source code changes in a given snapshot,
- a source code fragment not belonging to the entity changes together with a given entity, or
- an existing entity disappears in a given snapshot.

As Figure 2 shows, we can make these determinations because an entity, identified in a given snapshot *i*, is trackable forward and backward by



**Figure 1. Differencing algorithm. Step 1: The algorithm detects the unchanged, deleted, and added lines using diff. Step 2: It computes hunk similarity (with cosine similarity) to trace hunks across releases. Step 3: It uses line similarity (Levenshtein) to identify changed lines. Finally, the algorithm iterates Steps 2 and 3 on the remaining hunks.**

following its changed and unchanged lines. Specifically, we can get information about whether the entity changed in another snapshot  $j \neq i$ .

### Line-Differencing Algorithm Performance

We compared ldiff's performance with that of the widely adopted Unix diff. First, we assessed

**Table 1**  
**Similarity metrics**

Set-based metric	Definition
Dice( $X, Y$ )	The ratio between twice the intersection of $X$ and $Y$ and the sum of $X$ and $Y$ modules
Cosine( $X, Y$ )	The cosine of the angle between $X$ and $Y$ represented as vectors of a Euclidean space
Jaccard( $X, Y$ )	The fraction of common items ( $ X \cap Y $ ) with respect to overall items ( $ X \cup Y $ )
Overlap( $X, Y$ )	1 if the set $X$ is a subset of $Y$ or the converse; 0 if there is no overlap; $< 1$ otherwise
Sequence-based metric	Definition
Levenstein( $X, Y$ )	Measures the minimum edit distance that transforms $X$ into $Y$ in terms of add, delete, and substitute operations
Jaro( $X, Y$ )	Measures typical spelling deviations

ldiff's ability to identify moved line blocks and thus its ability to track a software entity when its position in a file changes. To this end, we randomly generated new releases of 100 source code files selected from two open source projects (PostgreSQL and openSSH) by randomly moving code fragments within the source code file. The fragments varied from 1 line to a maximum of 1/10 of the total number of lines. We assessed the algorithm in terms of precision and recall:

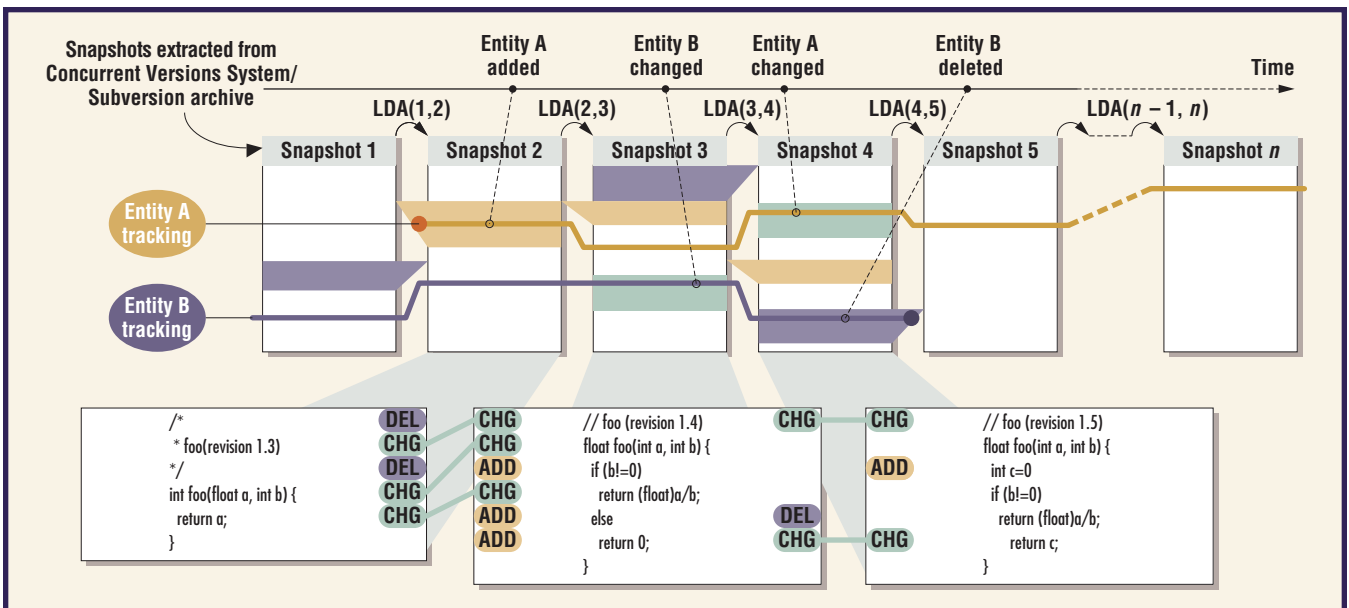
■ *precision* = number of correctly detected moves /

number of detected moves.

■ *recall* = number of correctly detected moves / number of generated moves.

As Figure 3a shows, the algorithm reveals a median precision of 92 percent and the recall increasing with the number of iterations, from 62 percent with one iteration to 73 percent with four iterations. Whereas the precision remains almost constant across iterations (it increases 0.7 percent from the first to the fourth iteration), the recall increases by 21 percent from the first to the fourth iteration. This difference is marginally significant:  $p$ -value = 0.05 computed using a one-tailed (because we're expecting improvements over subsequent steps) Mann-Whitney test.

The second assessment aimed to evaluate the ldiff accuracy in identifying changed, added, deleted, and unchanged source code lines by classifying changes in 11 change sets. We randomly extracted change sets from the ArgoUML Concurrent Versions System (CVS) repository, representing different types of changes, such as bug fixing, refactoring, or enhancement. We assessed the tool's precision by manually identifying false positives in classifications the algorithm made. The 11 change sets affected from 11 to 72 files (median = 19) and from 32 to 401 lines (median = 42). Figure 3b shows the median ldiff and Unix diff accuracy and the interquartile range (between the third and first quartile). (For the ldiff syntax, see the "Ldiff: A Support Tool" sidebar.)



**Figure 2. Tracking source code entities across subsequent system snapshots. The proposed approach enables locating a source code entity in subsequent code snapshots. It allows for identifying when a developer adds, deletes, or changes a source code line across subsequent snapshots.**



We tested the results' significance by using a two-tailed (because we don't know a priori whether *ldiff* performs better than the Unix *diff*) Mann-Whitney test. *ldiff* outperforms the Unix *diff* in identifying changed lines ( $p$ -value  $< 0.001$ ), whereas the Unix *diff* performs better in identifying added and deleted lines ( $p$ -value = 0.009 and  $< 0.0001$ , respectively). The Unix *diff* does better in the latter results because *ldiff* classifies added and deleted lines as potential changed lines, generating both false negatives and false positives. Because *ldiff* relies on *diff* in identifying unchanged lines, we found no difference in those results, which had an average precision of 99 percent.

We empirically evaluated the time complexity by executing the algorithm with different hunk sizes. Results show that the execution time grows quadratically ( $R^2$ -adj = 92.3 percent) with the number of evaluated line pairs in each hunk. For example, on a 2-GHz Intel Centrino laptop with 1 Gbyte of RAM, one algorithm iteration takes two seconds to classify 34 line pairs and 54 seconds to classify 171 line pairs.

The raw data we used on all of these experiments are available for replication at <http://rcost.unisannio.it/cerulo/ldiff-rawdata.tgz>.

## Application Examples

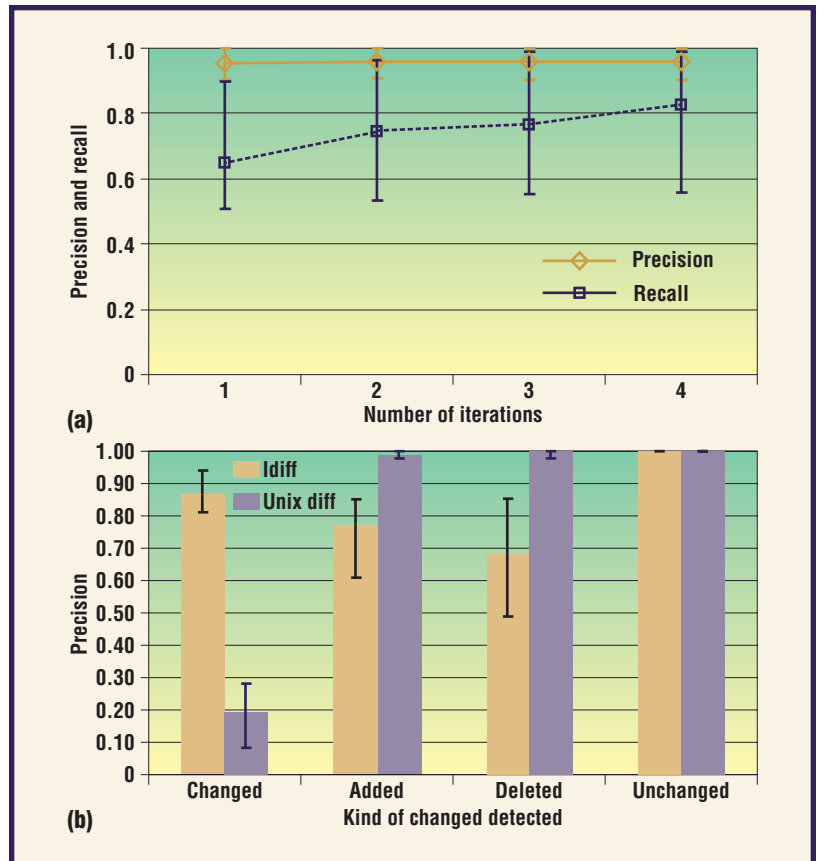
Our proposed approach is useful in a variety of applications. The two examples we discuss in this section are from a set of empirical studies that we performed. Errors that the entity (clone or vulnerability) detection and *ldiff* tools introduced affected the results.

## Tracking Source Code Clones

In the past, source code clones were often considered bad software development practice because they can potentially cause maintainability problems owing to the need to propagate changes over them. However, recent studies have shown that clones aren't necessarily a bad thing.<sup>5,6</sup> In many cases, developers have used cloning as a development practice.

We applied our code-tracking approach to analyze change propagation across clones as the Bauhaus *ccdiml* tool ([www.bauhaus-stuttgart.de/bauhaus/index-english.html](http://www.bauhaus-stuttgart.de/bauhaus/index-english.html)) detected them. Specifically, we classified cases in which

- changes are consistently propagated (within the same change set) to all clone fragments belonging to the same clone class;
- changes are propagated with some delay—for example, one clone fragment is modified in a



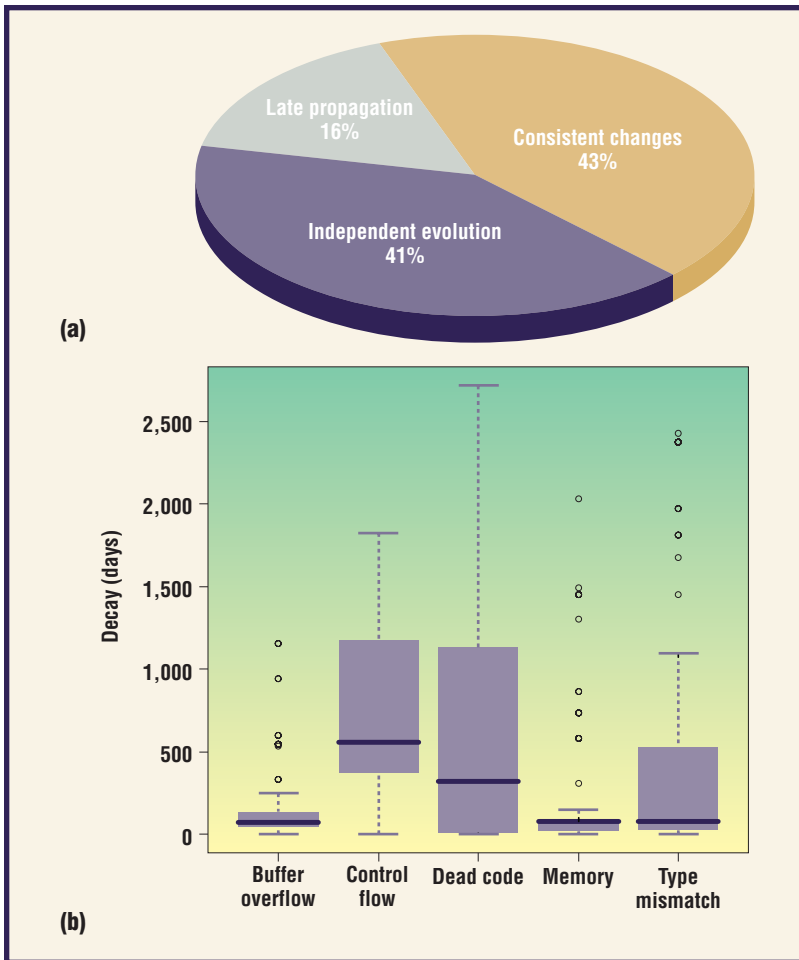
**Figure 3. Line-differencing performance evaluation. (a) Measuring the ability to identify moved lines on a set of 100 source code files showed a marginally significant difference in recall (*ldiff* parameters:  $LT = 1$ ,  $HT = \text{top } 3$ ,  $i = 1 \dots 4$ ). (b) Measuring the precision of *ldiff* and Unix *diff* to identify changed, added, deleted, and unchanged lines showed strengths for each approach (*ldiff* parameters:  $LT = 0.4$ ,  $HT = \text{top } 3$ ,  $i = 1$ ).**

change set and another fragment undergoes the same change in a later change set; and

- clones evolve independently—for example, to implement different features.

As Figure 4a (on the next page) shows, the percentage of late propagations is, indeed, low (16 percent), at least for the reported case study (PostgreSQL; [www.postgresql.org](http://www.postgresql.org)). Most of the clones either change consistently (43 percent) or evolve independently (41 percent). Results for other case studies (such as ArgoUML; <http://argouml.tigris.org>) indicate even lower late-propagation rates (3 percent) and a majority of consistent changes (61 percent).

The *ldiff* tool's ability to track clone evolution makes it suitable for use in implementing recommender systems that can automatically track clone change propagations and warn developers of improperly propagated changes. This approach can be an alternative to asking developers to explicitly label clone fragments (detected through



**Figure 4. Source code clone tracking. (a) The percentage of clone fragments that Bauhaus ccdiml in PostgreSQL detected as undergoing consistent changes, independent evolution, or late propagation. (b) Box plots of the decay for vulnerable instructions that Splint detected in the Squid project.**


clone-detection tools) for tracking.<sup>7</sup> Such a solution could help avoid, for example, the same bug appearing twice in the system. This was the case with PostgreSQL, in which a source code fragment underwent a bug fixing. Developers discovered the same bug six months later because the change hadn't been correctly propagated. The developer who committed the second change wrote in the CVS note, "I had previously fixed the identical bug in `oper_select_candidate` but didn't realize that the same error was repeated over here."

### Monitoring Vulnerable Instructions

Avoiding security attacks is crucial when developing network applications. Attacks, such as buffer overflows and cross-site scripting, are more and more frequent, causing unauthorized access to systems and data or denials of service. Static-analysis tools such as Splint ([www.splint.org](http://www.splint.org)) enable detec-

tion of instructions that could potentially cause security attacks.

In addition to simply detecting vulnerable instructions, analyzing how the developers maintain them over time would also be valuable, tracking changes from introduction until they disappear from the system. In particular, it's possible to compare the decay time, or the total time a vulnerability is in the system. This process is similar to Sung-hun Kim and Michael Ernst's method for studying how developers fixed warnings that compilers produced.<sup>8</sup> Figure 4b compares the decay time of different types of vulnerabilities Splint detected in the Squid Web proxy ([www.squid-cache.org](http://www.squid-cache.org)) source code. It further indicates how developers removed vulnerabilities such as buffer overflows and memory allocation problems more quickly than others. Also possible is modeling the vulnerability decay by means of a probability distribution or estimating the likelihood that removing a vulnerability is necessary.<sup>9</sup> We found that for some vulnerability categories (such as buffer overflows), the likelihood a vulnerability must disappear from the system decreases exponentially with time.

**W**e have shown how ldiff is able to overcome the Unix diff's limitations to identify changed text lines, and how it can be used in the context of software evolution studies—for example, to track the evolution of source code clones or monitor vulnerable instructions. Future work aims at integrating ldiff in an IDE and realizing an ldiff front end able to visually trace the evolution of software artifacts. 

### References

1. G. Canfora, L. Cerulo, and M. Di Penta, "Identifying Changed Source Code Lines from Version Repositories," *Proc. 4th Int'l Workshop Mining Software Repositories (MSR 07)*, IEEE CS Press, 2007, pp. 14–22.
2. B. Fluri et al., "Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction," *IEEE Trans. Software Eng.*, vol. 33, no. 11, 2007, pp. 725–743.
3. H. Gall, M. Jazayeri, and J. Krajewski, "CVS Release History Data for Detecting Logical Couplings," *Proc. 6th Int'l Workshop Principles of Software Evolution (IWVSE 03)*, IEEE CS Press, 2003, pp. 13–23.
4. T. Zimmermann et al., "Mining Version Histories to Guide Software Changes," *Proc. 26th Int'l Conf. Software Eng. (ICSE 04)*, IEEE CS Press, 2004, pp. 563–572.
5. C. Kapsner and M.W. Godfrey, "'Cloning Considered Harmful' Considered Harmful," *Proc. 13th Working Conf. Reverse Eng.*, IEEE CS Press, 2006, pp. 19–28.

6. M. Kim et al., "An Empirical Study of Code Clone Genealogies," *Proc. European Software Eng. Conf. and the ACM Symp. Foundations of Software Eng.*, ACM Press, 2005, pp. 187–196.
7. E. Duala-Ekoko and M.P. Robillard, "Tracking Code Clones in Evolving Software," *Proc. 29th Int'l Conf. Software Eng. (ICSE 07)*, IEEE CS Press, 2007, pp. 158–167.
8. S. Kim and M.D. Ernst, "Which Warnings Should I Fix First?" *Proc. 6th Joint Meeting European Software Eng. Conf. and the ACM SIGSOFT Int'l Symp. Foundations of Software Eng.*, ACM Press, 2007, pp. 45–54.
9. M. Di Penta, L. Cerulo, and L. Aversano, "The Evolution and Decay of Statically Detected Source Code Vulnerabilities," *Proc. 8th IEEE Working Conf. Source Code Analysis and Manipulation (SCAM 08)*, IEEE CS Press, 2008, pp. 101–110.

## About the Authors



**Gerardo Canfora** is a full professor in the University of Sannio Department of Engineering. His research interests include service-centric software engineering, software maintenance, and empirical software engineering. Canfora received a laurea degree in electronic engineering from the University of Naples. He's a member of the IEEE Computer Society. Contact him at [canfora@unisannio.it](mailto:canfora@unisannio.it).

**Luigi Cerulo** is a postdoctoral researcher in the University of Sannio Department of Engineering. His research interests include mining software repositories, software maintenance, and empirical software engineering. Cerulo received a PhD in computer engineering from the University of Sannio. He's a member of the ACM. Contact him at [lcerulo@unisannio.it](mailto:lcerulo@unisannio.it).



**Massimiliano Di Penta** is an assistant professor in the University of Sannio Department of Engineering. His research interests include empirical software engineering, software maintenance, and search-based software engineering. Di Penta received a PhD in computer engineering from the University of Sannio. He's a member of the IEEE, the IEEE Computer Society, and the ACM. Contact him at [dipenta@unisannio.it](mailto:dipenta@unisannio.it).

For more information on this or any other computing topic, please visit our Digital Library at [www.computer.org/csdl](http://www.computer.org/csdl).

### Advertising Information January/February 2009 IEEE Software

Advertiser	Page
<b>ICSM 2009</b>	<b>Cover 2</b>
<b>Nu Info Systems, Inc.</b>	<b>8</b>
<b>SD West 2009</b>	<b>Cover 3</b>
<b>Seapine Software Inc.</b>	<b>Cover 4</b>
<b>WICSA 2009</b>	<b>1</b>
<b>Classified Advertising</b>	<b>7</b>

#### Advertising Personnel

Marion Delaney  
IEEE Media, Advertising Dir.  
Phone: +1 415 863 4717  
Email: [md.ieeemedia@ieee.org](mailto:md.ieeemedia@ieee.org)

Marian Anderson  
Sr. Advertising Coordinator  
Phone: +1 714 821 8380  
Fax: +1 714 821 4010  
Email: [manderson@computer.org](mailto:manderson@computer.org)

Sandy Brown  
Sr. Business Development Mgr.  
Phone: +1 714 821 8380  
Fax: +1 714 821 4010  
Email: [sb.ieeemedia@ieee.org](mailto:sb.ieeemedia@ieee.org)

### Advertising Sales Representatives

#### Recruitment:

Mid Atlantic  
Lisa Rinaldo  
Phone: +1 732 772 0160  
Fax: +1 732 772 0164  
Email: [lr.ieeemedia@ieee.org](mailto:lr.ieeemedia@ieee.org)

New England  
John Restchack  
Phone: +1 212 419 7578  
Fax: +1 212 419 7589  
Email: [j.restchack@ieee.org](mailto:j.restchack@ieee.org)

Southeast  
Thomas M. Flynn  
Phone: +1 770 645 2944  
Fax: +1 770 993 4423  
Email: [flyntom@mindspring.com](mailto:flyntom@mindspring.com)

Midwest/Southwest  
Darcy Giovingo  
Phone: +1 847 498-4520  
Fax: +1 847 498-5911  
Email: [dg.ieeemedia@ieee.org](mailto:dg.ieeemedia@ieee.org)

Northwest/Southern CA  
Tim Matteson  
Phone: +1 310 836 4064  
Fax: +1 310 836 4067  
Email: [tm.ieeemedia@ieee.org](mailto:tm.ieeemedia@ieee.org)

Japan  
Tim Matteson  
Phone: +1 310 836 4064  
Fax: +1 310 836 4067  
Email: [tm.ieeemedia@ieee.org](mailto:tm.ieeemedia@ieee.org)

Europe  
Hilary Turnbull  
Phone: +44 1875 825700  
Fax: +44 1875 825701  
Email: [impress@impressmedia.com](mailto:impress@impressmedia.com)

#### Product:

US East  
Joseph M. Donnelly  
Phone: +1 732 526 7119  
Email: [jdonnelly@briggsdonnelly.com](mailto:jdonnelly@briggsdonnelly.com)

US Central  
Darcy Giovingo  
Phone: +1 847 498-4520  
Fax: +1 847 498-5911  
Email: [dg.ieeemedia@ieee.org](mailto:dg.ieeemedia@ieee.org)

US West  
Lynne Stickrod  
Phone: +1 415 503 3936  
Fax: +1 415 503 3937  
Email: [l.stickrod@att.net](mailto:l.stickrod@att.net)

Europe  
Sven Anacker  
Phone: +49 202 27169 11  
Fax: +49 202 27169 20  
Email: [sanacker@intermediapartners.de](mailto:sanacker@intermediapartners.de)