Tracking Your Changes: a Language-Independent Approach

Gerardo Canfora, Luigi Cerulo, Massimiliano Di Penta
RCOST – Department of Engineering – University of Sannio, Benevento, Italy
canfora@unisannio.it, lcerulo@unisannio.it, dipenta@unisannio.it

Abstract

The availability of powerful differencing algorithms is crucial to track the evolution of source code, for example with the purpose of monitoring clones or vulnerable statements. In this paper we present a language-independent approach to track the evolution of code fragments, based on a novel differencing algorithm, that overcomes limitations of the Unix diff. We show how the algorithm is able to track the evolution of code elements in real-world software systems with acceptable precision, and provide examples—such as clone tracking and vulnerability tracking—where the algorithm has been successfully applied.

Keywords: differencing tools, mining software archives, software evolution.

1 Introduction

Versioning systems and bug tracking systems represent valuable assets to support large software projects, where many developers spread around the world, and running around the clock, participate, and many users report bugs or proposals for enhancement. In addition to supporting development activities, versioning systems are a precious source of information that can be exploited to study, or monitor, the evolution of a software system over time. To this aim, it is necessary to track source code artifacts across file revisions. In versioning systems, this is done using the Unix *diff* algorithm, that, to preserve space, treats any change as the minimum sequence of additions and deletions. This limits the ability of *diff* to distinguish line additions and removal from line changes. If a source code line is changed, the versioning system records it as the removal of the old line and the addition of the new one. This makes it difficult to know whether the modification consisted of a complete replacement of the line with a new one, or whether, instead, the old line was only partially modified. There is a degree of subjectiveness in doing such a classification, in that also humans can disagree on how something has been changed.

Although there is no objective way to determine whether a source code line has been changed, or whether it has been replaced by a new line, differencing algorithms that makes such a classification with a reasonable margin of error are foreseen.

We present an approach that allows for tracking software entities across multiple revisions of a file. The approach relies on a language independent differencing algorithm we recently introduced [1]. This algorithm overtakes the above mentioned limitations of *diff* and, differently from algorithms based on the code Abstract Syntax Tree (AST) [4], it does not require the code to be parsed. This makes the algorithm suitable—whenever a detailed classification of the change is not needed—to track the evolution of any software entity—source code or not—that can be treated as a sequence of lines, including, for example, requirements, use cases, or test cases.

The differencing algorithm is implemented in a tool named *ldiff* ("line diff(erencing)", see Box A), having syntax and output similar to the Unix *diff*. This makes the tool immediately usable by *diff* users, as well as its integration in Integrated Development Environments (IDEs) or recommender systems relying on a Unix *diff* like algorithm.

2 Software entity tracking process

In this Section we describe a process to track, within a file, the evolution of software entities one is interested to observe. A software entity is intended as a set of (not necessarily adjacent) lines from a textual artifact, for example, a source code clone, a source code line containing a vulnerable statement, or a comment. The process comprises four stages aimed at (i) identifying the units of analysis, (ii) identifying the entities to be tracked, (iii) tracking the entities, and (iv) identifying entity changes.

2.1 Stage A: Extract software system snapshots at a level of granularity able to capture the developer activities

The evolution of a software entity can be studied across system releases, single file commits, or by grouping together logically related commits. As proposed by Gall *et al.* [5], the evolution of a software system can be seen as a sequence of *Snapshots* generated by *Change Sets*, representing the logical changes performed by a developer in terms of added, deleted, and changed source code lines. Such sequence can be reconstructed with various approaches. For example, the Zimmermann *et al.* approach [13]—which we used—considers the sequences of file revisions that share the same author, branch, and commit notes, and such that the difference between the timestamps of two subsequent commits is less or equal than 200 seconds.

2.2 Stage B: Identify entities of interest

Before tracking the evolution of an entity of interest, we need to identify it in the source code (or textual file). This can be done manually—e.g., the developer identifies a source code entity s/he wants to keep track on—or automatically, as it happens for clones or vulnerabilities (see Section 4). In any case, the entity is identified as a set of textual/source code file lines, and the next stages will be used to track the changes of these lines.

2.3 Stage C: Track source code entity changes

The core of the proposed tracking approach is a Line Differencing Algorithm (LDA). A 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 only provide information at the line level as follows:

- the line was left *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}.

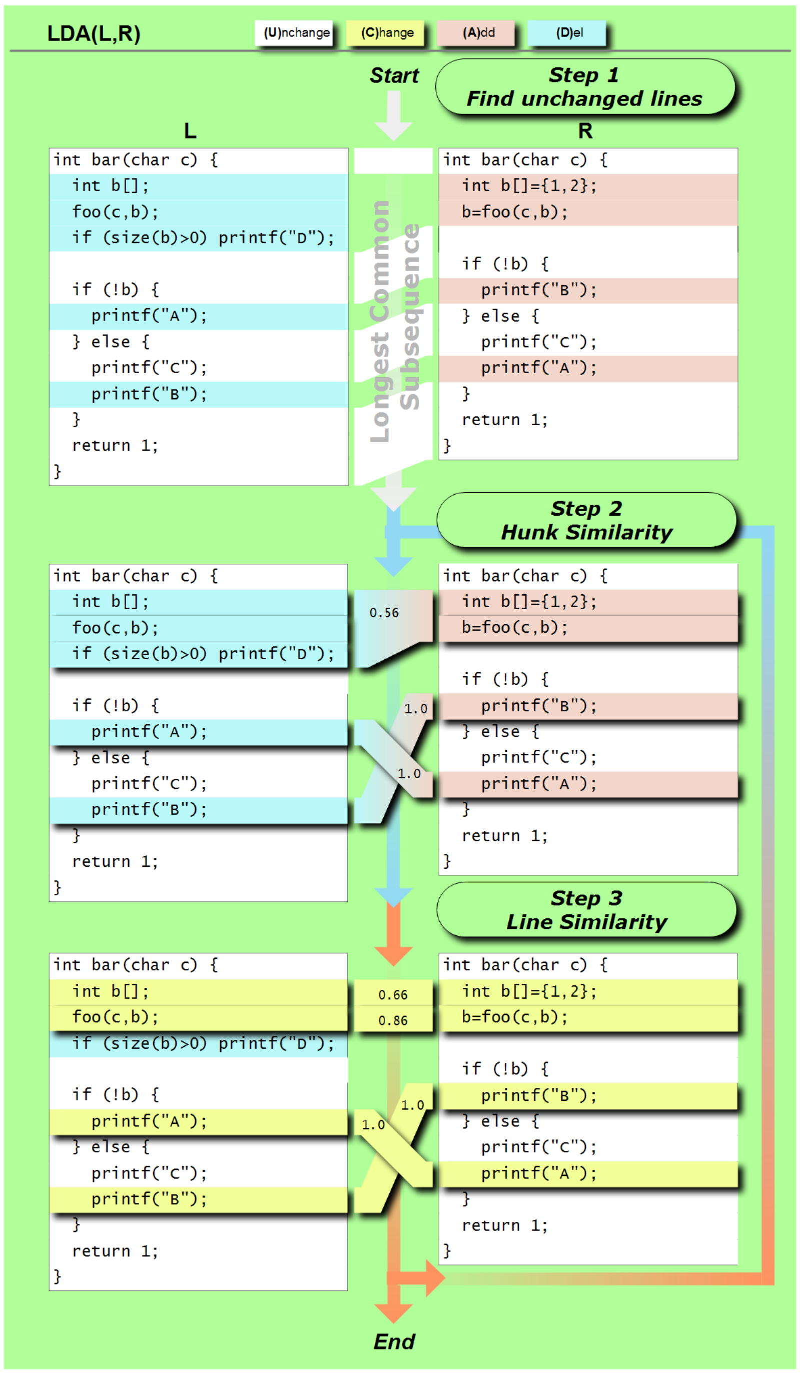
One of the most widely known LDA algorithms is the Unix *diff*, which however is not able to detect *changed* lines, but treats them as sequence of *deleted* and *added* lines. When *diff* finds a sequence of additions and deletions, starting from the same position in the file, then it assumes that the block has been *changed*. However, this may not be the case, since the block—or part of it—could have been replaced by a completely different one. If we consider, for example, source code lines 2–4 in the top-left (L) fragment of Figure 1, changing as shown in the top-right (R) fragment, 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);
```

indicating that the block composed of three lines has been changed into a block of two lines. However, this is very likely not what the programmer did; instead, the third line has been removed and the first and second have been changed, respectively into the first and second of the second block. As mentioned before, no automatic differencing tool would be able to unambiguously distinguish changes from addition and removal as made by the programmer, but a tool able to suggest likely changes would be desirable.

To this aim, we have proposed an LDA particularly suitable for code tracking purposes [1]. Let us consider the topmost-left source code fragment in Figure 1, evolving as shown on the right side. The algorithm starts (Step 1) by applying the Unix *diff* with the purpose of identifying unchanged lines. Of course, the question whether the unchanged lines—as detected by *diff*—are a good approximation of the set of actually unchanged lines, arises. The Unix *diff* searches, among many possible common subsequences between the two file releases, the longest ones. This reflects the assumption that a programmer tends to minimize the change effort by reusing existing lines, although this may or may not be the actual intention of the programmer (see Section 3).

Step 2 compares the *hunks*—i.e., sequences of adjacent lines—of L and R that have not been classified as unchanged (colored in cyan and magenta respectively). Specifically, the comparison is made between all possible pairs of L and R hunks using a hunk similarity measure. Then (Step 3), the algorithm takes the topmost H distinct hunk pairs as shown in the middle of Figure 1 and, for each pair, it performs a *line-by-line* comparison. In other words, the algorithm compares lines of the left-side hunk with lines of the right-side by using a line similarity measure; the line pairs having a similarity above a given threshold L are classified as *changed* lines (yellow lines in the bottom part of Figure 1). All sets of adjacent lines not classified as *changed* are considered as new *hunks* for a subsequent iteration of Steps 2 and 3. The tool can be calibrated by selecting the maximum number of iterations, with the aim of



```
int bar(char c) {
  int b[];
  foo(c,b);
  if (size(b)>0) printf("D");

  if (!b) {
    printf("A");
  } else {
    printf("C");
    printf("B");
  }
  return 1;
}
```

```
int bar(char c) {
  int b[]={1,2};
  b=foo(c,b);

  if (!b) {
    printf("B");
  } else {
    printf("C");
    printf("A");
  }
  return 1;
}
```

Figure 1. Differencing algorithm. Step 1: unchanged lines (white), deleted (cyan) and added (red) are detected using *diff*. Step 2: hunk similarity is computed (with cosine similarity) to trace hunks across releases. Step 3: line similarity (Levenshtein) is used to identify changed lines (yellow). Then, Steps 2-3 are iterated on the remaining hunks (cyan/magenta).

Table 1. Similarity metrics. Set-based measures are used to compute hunk similarity (Step 2 of Figure 1), while sequence-based measures are used to compute line similarity (Step 3 of Figure 1).

SET BASED	
$Dice(X, Y)$	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 an 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 (also known as distances)	
$Levensthein(X, Y)$	Measures the minimum edit distance which transforms X into Y in terms of add, del, and substitute operations
$Jaro(X, Y)$	Measures typical spelling deviations

increasing the recall (see Section 3). This because subsequent iterations will consider combinations of hunks previously discarded. Among other cases, this would be useful to detect merge and split, where each iteration will only match one source (in the case of merge) or one target (in the case of split). As shown in Box A, H, L, 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 (e.g., characters, words, or tokens). Set-based metrics, i.e., metrics that do not consider ordering information, are suited to compute hunk similarity. Instead, sequence based metrics—i.e., taking into account the order in which items appear—are suited for line similarity. Table 1 shows examples of metrics used to compute these similarities. In the studies and examples reported in this paper, we always use the Cosine set based metric on words, and the Levensthein sequence based metric on characters.

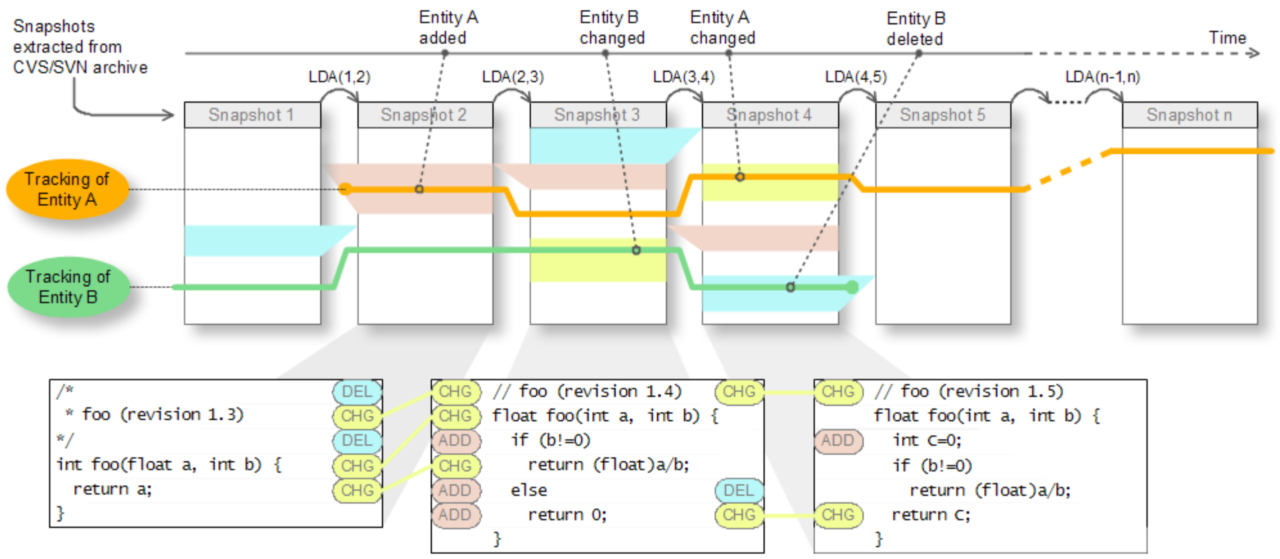
2.4 Stage D: Identify changes occurred in software entities

By relying on the information obtained in the previous steps, we are able to determine (see Figure 2): (i) if a new software entity of interest appears in a given snapshot; (ii) if the source code of an entity changes in a given snapshot; (iii) if a source code fragment not belonging to the entity changes together with a given entity; or (iv) if an existing entity disappears in a given snapshot.

As shown in Figure 2, this can be done because an entity, identified in a given snapshot i, can be tracked forward and backward by following its changed and unchanged lines, getting information about whether or not it was changed in another snapshot $j \neq i$.

3 Performances of the Line Differencing Algorithm

We compared performances of our LDA algorithm with those of the widely-adopted Unix *diff*. First, we assess the ability of the LDA algorithm to identify moved line blocks, and thus its ability to track a software entity when its position in a file changes. To this aim, we randomly generate new releases of 100 source code files selected from two open source projects (PostgreSQL, openSSH), by randomly moving code fragments—varying from 1 line to a maximum of 1/10 of the total number of lines—within the source code file. We assessed the algorithm in terms of *precision* and *recall*:



```
/*
 * foo (revision 1.3)
*/
int foo(float a, int b) {
  return a;
}
```

```
// foo (revision 1.4)
float foo(int a, int b) {
  if (b!=0)
    return (float)a/b;
  else
    return 0;
}
```

```
// foo (revision 1.5)
float foo(int a, int b) {
  int c=0;
  if (b!=0)
    return (float)a/b;
  return c;
}
```

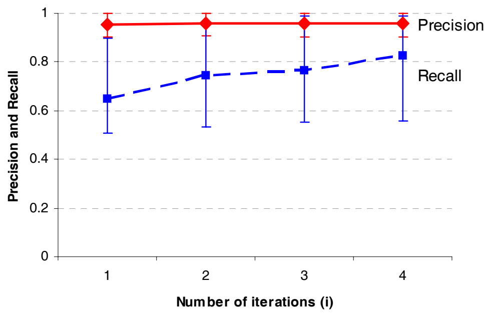
Figure 2. Tracking source code entities across subsequent system snapshots. The proposed approach makes it possible to locate a source code entity in subsequent code snapshots. Also, it allows for identifying when a source code line is added, deleted, or changed across subsequent snapshots.

$$precision = \frac{\#\ of\ correctly\ detected\ moves}{\#\ of\ detected\ moves} \qquad recall = \frac{\#of\ correctly\ detected\ moves}{\#\ of\ generated\ moves}$$

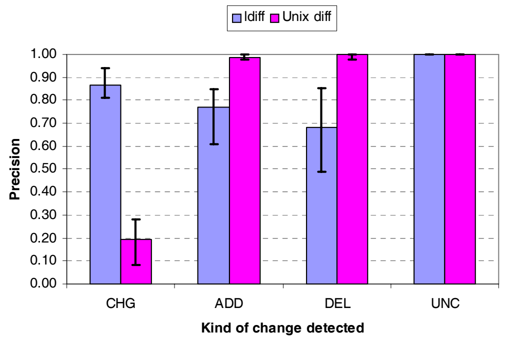
As Figure 3-a shows, the algorithm reveals a median precision of 92% and the recall increasing with the number of iterations, from 62% with one iteration to 73% with four iterations. While the precision remains almost constant across iterations (it increases of 0.7% from the first to the fourth iteration), the recall increases of 21% from the first to the fourth iteration, and the difference is marginally significant (p-value=0.05 computed using a one-tailed[1] Mann-Whitney test).

The second assessment aims at evaluating the LDA algorithm accuracy in the identification of changed, added, deleted, and unchanged source code lines, by classifying changes occurred into 11 change sets randomly extracted from the ArgoUML CVS (Concurrent Versions Systems) repository and representative of different kinds of changes, i.e., bug fixing, refactoring, or enhancement. We assessed the tool precision by manually identifying false positives occurred in classifications made by the algorithm. The 11 change sets affected between 11 and 72 files (median=19), and between 32 and 401 lines (median=42). Figure 3-b shows the median LDA and Unix *diff* accuracy over the experiments performed, and the interquartile range (between the third and first quartile). We tested the significance of the obtained results, by using a two-tailed[2] Mann-Whitney test. The LDA algorithm exhibits significantly higher performances in the identification of changed lines (p-value$<$0.001), while the Unix *diff* performs better in the

[1]Since we are expecting improvements over subsequent steps.
[2]Since we do not know a-priori whether *ldiff* performs better than the Unix *diff*.



(a)



(b)

Figure 3. Line differencing performance evaluation. (a) Ability to identify moved lines on a set of 100 source code files. The figure shows the median precision and recall—and the inter-quartile difference—for different numbers of iterations (ldiff parameters: L=1, H=top 3, i=1 . . . 4). (b) Precision of ldiff and Unix diff to identify changed, added, deleted, and unchanged lines. Results are based on 11 change sets randomly extracted from the PostgreSQL CVS repository (ldiff parameters: L=0.4, H=top 3, i=1). For the ldiff syntax, see Box A.

identification of added and deleted lines (p-value=0.009 and $<$0.0001 respectively) because *ldiff* classifies added and deleted lines as potential changed lines, causing an increment of both false negatives and positives. There is no difference in the identification of the unchanged lines—for which, in this experiment, we found an average precision of 99%—as *ldiff* relies on *diff* for that step.

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

Raw data used on all the above experiments are available for replication purposes[3].

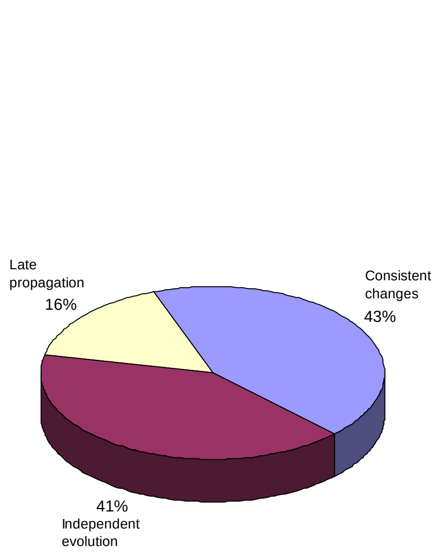
4 Examples of Applications

The proposed approach can be used for a variety of applications. In the following we report two examples of application taken from a set of empirical studies we performed. Results are affected by errors introduced by the entity (clone or vulnerability) detection tool and by the *ldiff* tool itself.

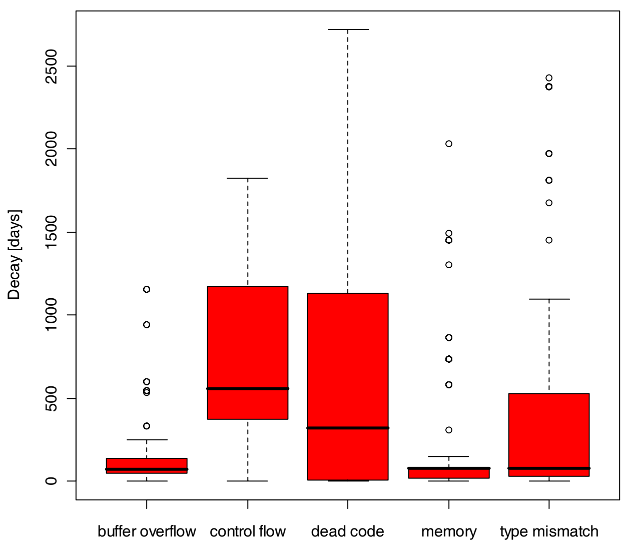
4.1 Are source code clones maintained consistently?

In the past, source code clones have been often considered as a bad software development practice, since they can potentially cause maintainability problems, due to the need for propagating changes over them. Recent studies [7, 9] have shown that clones are not necessarily a bad thing, as in many cases cloning has been used as a development practice.

[3]http://rcost.unisannio.it/cerulo/ldiff-rawdata.tgz



(a) Clone change propagation



(b) Vulnerability Decay

Figure 4. (a) Percentage of clone fragments, detected by Bauhaus *ccdiml* in PostgreSQL, that underwent to consistent changes, independent evolution, or late propagation. (b) Box plots of the decay for vulnerable instructions detected by Splint in the Squid project.

We applied the proposed code tracking approach to analyze change propagation across clones as they were detected by Bauhaus *ccdiml* tool[4]. Specifically, we classified cases where (i) changes are consistently propagated —within the same change set— to all clone fragments belonging to the same clone class; (ii) changes are propagated with some delay, e.g., one clone fragment is modified in a change set and another fragment undergoes the same change, in a later change set; and (iii) clones evolve independently, e.g., to implement different features. As Figure 4-a shows, the percentage of late propagations is, indeed, low (16%) at least for the reported case study (PostgreSQL[5]), while most of the clones are either consistently changed (43%) or evolve independently (41%). Results for other case studies (e.g., ArgoUML[6]) indicate even lower late propagation rates (3%) and a majority of consistent changes (61%).

The ability of the *ldiff* tool to track the evolution of clones makes it suitable to be used to implement recommender systems able to automatically keep track of clone change propagations, and to warn developers when a change is not properly propagated. This can be an alternative to ask developers to explicitly label clone fragments—detected by means of clone detection tools—to keep track of them [3], and would be useful to avoid, for example, the same bug to appear twice in the system. It was the case

[4]http://www.bauhaus-stuttgart.de/bauhaus/index-english.html
[5]http://www.postgresql.org/
[6]http://argouml.tigris.org/

of PostgreSQL, where a source code fragment underwent a bug fixing, and the same bug was discovered six months later, because the change was not 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..."*.

4.2 Do developers take care of potentially vulnerable instructions?

Avoiding security attacks is crucial when developing network applications, since attacks such as buffer overflows and cross-site scripting are more and more frequent, causing unauthorized access to system and data, or denial of services. Static analysis tools such as *Splint*[7] allow for detecting instructions that could potentially cause security attacks.

Other than just detecting vulnerable instructions, it would be desirable to analyze how they are maintained over time, by tracking their changes from their introduction until they disappear from the system. In particular, it is possible to compare the decay time—i.e., the time a vulnerability remains in the system from its introduction until its removal, similarly to what did by Kim and Ernst [10] to study how developers fixed warnings produced by compilers. Figure 4-b compares the decay time of different kinds of vulnerabilities detected by *Splint* in the Squid Web proxy[8] source code, and indicates how vulnerabilities such as buffer overflows and memory allocation problems are removed quicker than others. In addition, it would be possible to model the vulnerability decay by means of a probability distribution, or to estimate the likelihood a vulnerability has to be removed [2]: we found that for some vulnerability categories (e.g., buffer overflows) the likelihood a vulnerability has to disappear from the system exponentially decreases with the time.

5 Conclusions

We have presented an approach that allows fine grained analysis of software artifacts evolution. The approach is based on a line-differencing algorithm, which enhances the Unix *diff* by computing set similarity over line hunks, and sequence similarity between lines. With respect to alternative approaches, it has several advantages: (i) it is better able than the Unix *diff* to identify and track changed lines; and (ii) differently from approaches based on AST differencing, it is language independent and can be applied to any kind of textual software artifact; On the other hand, it also has some limitations: (i) it is less precise than the Unix *diff* in the identification of added and deleted lines; and (ii) it is not able to perform a detailed identification of the kind of change occurred, as it can be done with AST differencing approaches.

Other than just using *ldiff* as an alternative to the Unix *diff*, we have shown how it can be used in the context of software evolution studies, for example aimed at tracking the evolution of source code clones, or at analyzing the presence of vulnerable instruction in a software system over the time.

References

[1] G. Canfora, L. Cerulo, and M. Di Penta. Identifying changed source code lines from version repositories. In *Fourth International Workshop on Mining Software Repositories, May 19-20, Minneapolis, MN, USA*, pages 14–22. IEEE Computer Society, 2007.

[7]http://www.splint.org/

[8]http://www.squid-cache.org/

[2] M. Di Penta, L. Cerulo, and L. Aversano. The evolution and decay of statically detected source code vulnerabilities. In *proceedings of the 8th IEEE Working Conference on Source Code Analysis and Manipulation*, 2008 (to appear).

[3] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pages 158–167, 2007.

[4] B. Fluri, M. Würsch, M. Pinzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Software Eng.*, 33(11):725–743, 2007.

[5] H. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. In *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, page 13. IEEE Computer Society, 2003.

[6] M. W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Trans. Software Eng.*, 31(2):166–181, 2005.

[7] C. Kapser and M. W. Godfrey. 'Cloning considered harmful' considered harmful. In *Proceedings of the 2006 Working Conference on Reverse Engineering*, Benevento, Italy, October 2006. IEEE Computer Society Press.

[8] M. Kim and D. Notkin. Program element matching for multi-version program analyses. In *Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR 2006, Shanghai, China, May 22-23, 2006*, pages 58–64, 2006.

[9] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *Proceedings of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering*, pages 187–196, Lisbon, Portogal, September 2005. ACM Press.

[10] S. Kim and M. D. Ernst. Which warnings should i fix first? In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, pages 45–54, 2007.

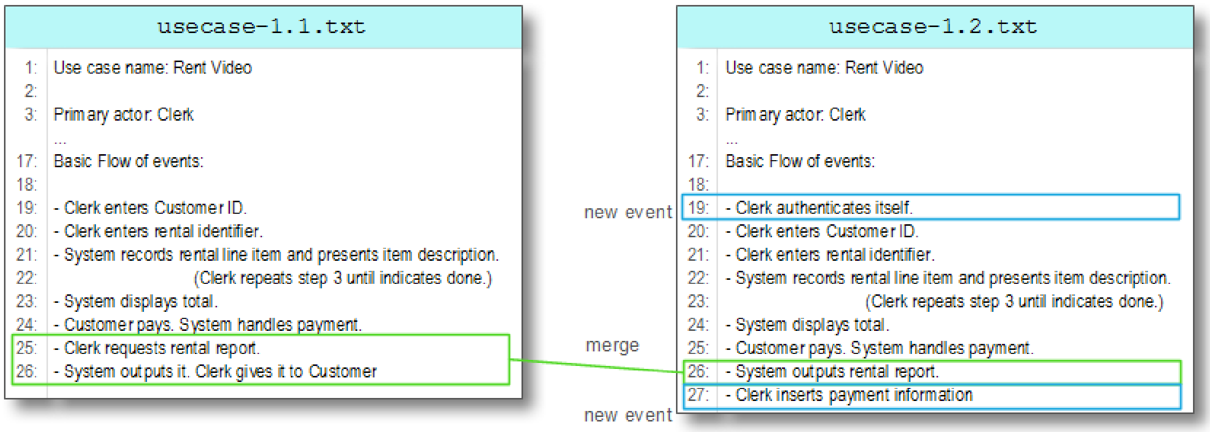
[11] S. P. Reiss. Tracking source locations. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 11–20, 2008.

[12] Z. Xing and E. Stroulia. Differencing logical UML models. *Autom. Softw. Eng.*, 14(2):215–259, 2007.

[13] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 563–572. IEEE Computer Society, 2004.

A Tool support (box)

A Perl implementation of the algorithm is available at the URL: *http://rcost.unisannio.it/cerulo/tools.html*. The tool supports a variety of hunk similarity metrics (Cosine, Jaccard, Dice, and Overlap), and different text item extraction techniques (chars, words, n–grams, C/C++/Java language tokens). Below we can see an example of using both *ldiff* and *diff* on two versions of a text artifact (use case):



usecase-1.1.txt

```
 1: Use case name: Rent Video
 2:
 3: Primary actor: Clerk
    ...
17: Basic Flow of events:
18:
19: - Clerk enters Customer ID.
20: - Clerk enters rental identifier.
21: - System records rental line item and presents item description.
22:               (Clerk repeats step 3 until indicates done.)
23: - System displays total.
24: - Customer pays. System handles payment.
25: - Clerk requests rental report.
26: - System outputs it. Clerk gives it to Customer
```

usecase-1.2.txt

```
 1: Use case name: Rent Video
 2:
 3: Primary actor: Clerk
    ...
17: Basic Flow of events:
18:
19: - Clerk authenticates itself.
20: - Clerk enters Customer ID.
21: - Clerk enters rental identifier.
22: - System records rental line item and presents item description.
23:               (Clerk repeats step 3 until indicates done.)
24: - System displays total.
25: - Customer pays. System handles payment.
26: - System outputs rental report.
27: - Clerk inserts payment information
```

ldiff is invoked with 1 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 (`-H 0.5:all`), a *char* line tokenizer (`-lt char`), a *Levensthein* line similarity measure (`-lm leven`), and a line similarity threshold of 0.4 (`-L 0.4`). The Unix *diff* command is invoked with its default parameters.

```
$ ldiff.pl -i 1 -ht word -hm cosine -H 0.5:all -lt char -lm leven -L 0.4 usecase-1.1.txt usecase-1.2.txt
18a19,19
> - Clerk authenticates itself.
25,25c26,26
< - Clerk requests rental report.
---
> - System outputs rental report.
25a27,27
> - Clerk inserts payment information
26,26d26
< - System outputs it.  Clerk gives it to Customer
```

```
$ diff usecase-1.1.txt usecase-1.2.txt
18a19
> - Clerk authenticates itself.
25,26c26,27
< - Clerk requests rental report.
< - System outputs it.  Clerk gives it to Customer
---
> - System outputs rental report.
> - Clerk inserts payment information
```

It can be noticed that the Unix *diff* is unable to detect the last added use case event, as it maps different adjacent lines into a single block change. Instead, *ldiff* considers the added event as an addition, and the merged events as a deletion, combined with a change.

B Related Work (box)

Kim and Notkin [8] classify code differencing algorithms, into algorithms working on a structured representation of the program (e.g., AST), and algorithms working on a flat representation (e.g., sequence of lines).

The work of Fluri *et al.* [4] belongs to the first class, and is able to detect, with a high precision, the nature of structural changes occurred in the source code, that cannot be identified by our line differencing algorithm. Besides the high value of change information that can be obtained from AST-based algorithms—e.g., identifying method signature changes, changes in class hierarchies, etc.—the inherent computational complexity may limit their application in large scale systems that underwent a high number of changes. In addition, a parser must be available and system snapshots must be analyzable with that parser, which may or may not be the case, since sometimes developers can leave the system in an incomplete or inconsistent state, e.g., with missing files, wrong links, etc.

Reiss [11] showed that source code can be tracked through multiple versions of a file by using relatively simple techniques, such as line matching based on the Levensthein distance.

Xing and Stroulia proposed *UMLDiff* [12] a differencing tool able to capture differences between UML models, identifying the addition, removal or change of elements such as methods, attributes, packages, etc. As for Fluri *et al.* [4], *UMLDiff* is more specific of *ldiff*, and more suited for performing detailed change analyses.

Godfrey and Zou [6] proposed an approach to detect merging and splitting of source code entities, considering characteristics—i.e., various metrics—of the entities, and callee/caller relationships between entities. Their approach is more specific to study refactoring activities—such as merging and splitting, while *ldiff* focuses on tracking sets of lines across file revisions and, above all, on distinguishing line changes from additions and deletions.

C Biographies

Gerardo Canfora is full professor at the University of Sannio, Dept. of Engineering. His research interests include service-centric software engineering, software maintenance, and empirical software engineering. He is member of the IEEE Computer Society.

Luigi Cerulo is post-doc at the University of Sannio, Dept. of Engineering. His research interests include mining software repositories, software maintenance, and empirical software engineering. He is member of the ACM.

Massimiliano Di Penta is assistant professor at the University of Sannio, Dept. of Engineering. His research interests include empirical software engineering, software maintenance, and search-based software engineering. He is member of the IEEE, IEEE Computer Society, and of the ACM.