Herodotos: A Tool to Expose Bugs' Lives

Nicolas Palix, Julia Lawall, Gilles Muller

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Herodotos: A Tool to Expose Bugs' Lives

Nicolas Palix — Julia Lawall — Gilles Muller

N° 6984

July 2009

Domaine 3

ISSN 0249-6399 ISRN INRIA/RR--6984--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
PARIS - ROCQUENCOURT

Herodotos: A Tool to Expose Bugs' Lives

Nicolas Palix* , Julia Lawall* , Gilles Muller†

Domaine 3 — Réseaux, systèmes et services, calcul distribué
Équipes-Projets Régal

Rapport de recherche n° 6984 — July 2009 — 16 pages

Abstract:

Software is continually evolving, to improve performance, correct errors, and add new features. Code modifications, however, inevitably lead to the introduction of defects. To prevent the introduction of defects, one has to understand why they occur. Thus, it is important to develop tools and practices that aid in defect finding, tracking and prevention.

In this paper, we propose a methodology and associated tool, Herodotos, to study defects over time. Herodotos semi-automatically tracks defects over multiple versions of a software project, independent of other changes in the source files. It builds a graphical history of each defect and gives some statistics based on the results. We have evaluated this approach on the history of a representative range of open source projects over the last three years. For each project, we explore several kinds of defects that have been found by static code analysis. We analyze the generated results to compare the selected software projects and defect kinds.

Key-words: History of defects and bugs, Software quality, Software evolution, Static code analysis

* DIKU, University of Copenhagen, Denmark {npalix,julia}@diku.dk
† gilles.muller@inria.fr

Centre de recherche INRIA Paris – Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex
Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

Herodotos: Un outil pour construire des historiques de fautes

Résumé :

Les logiciels évoluent continuellement afin d'améliorer les performances, corriger les erreurs, ou ajouter de nouvelles fonctionnalités. Cependant, les modifications du code conduisent inévitablement à l'introduction de défauts logiciels. Pour prévenir l'introduction de fautes, il est nécessaire de comprendre pourquoi elles surviennent. Il est donc important de développer des outils et des pratiques aidant à trouver, suivre et prévenir ces défauts.

Dans cet article, nous proposons une méthodologie et son outil associé, Herodotos, permettant d'étudier les fautes dans le temps. Herodotos suit, de manière semi-automatisée, les fautes logicielles sur plusieurs versions d'un même projet indépendamment des modifications annexes dans les fichiers sources. Il construit un historique graphique pour chaque faute et calcule des statistiques sur la vie des fautes. Nous avons évalué cette approche sur l'historique d'un ensemble de projets open source sur les trois dernières années. Pour chaque projet, nous explorons plusieurs types de fautes qui ont été trouvées par une analyse statique du code. Nous analysons les résultats produits pour comparer les projets sélectionnés et les types de fautes étudiées.

Mots-clés : Historique de défauts logiciels, Qualité logicielle, Évolution logicielle, Analyse statique de code

1 Introduction

Software is improving continually both by the addition of new features and the correction of bugs. For example, during the last 3 years, the Linux kernel has grown overall by 45%, and its drivers by more than 50%. Code modifications on this scale inevitably lead to the introduction of defects. To prevent the introduction of defects, one has to understand why they occur. Thus, it is important to develop tools and practices that aid in defect finding, tracking and prevention.

One approach that can help to better understand and thus prevent defects is to study defects in a software project over time. Such a study may answer questions such as: Does a project improve over time? How long does a defect remain in a software? Does the lifespan of a defect depend on the defect kind? Does it depend on the criticality of the software? Studying already seen defects of a project also gives important information about how to handle defect reports on the working version, *e.g.* by identifying which reports have already been determined to be false positives and can be safely ignored. In previous work, the information collected in a bug tracker has been used as a standard defect base for software projects. However, bug reports in such systems are often written in a natural language without explicit connection to the code base. It is thus very difficult to relate reported defects over successive versions or group them by bug/defect patterns. Moreover, as they are filled in by humans, the quantity and quality of defect reports depend on the habits of testers and users.

To obtain an accurate defect history, one must work at the code level. However, a challenge in studying the history of defects over time is that the code base may evolve around the defect site. For instance, over the last three years, there has been a new version of the Linux kernel every three months, and more than 6% of the Linux kernel has changed between every version [13, 14]. This implies that some defects will be added or removed and others will change position. To study defects over time, one has to first know where defects occur and thus have a consistent and substantial base of defects over multiple versions of the same software. Once a large base of defects is available, defects can be correlated from one software version to the next one even in the presence of nearby code modifications.

In this paper, we propose the methodology illustrated in Figure 1, and associated tool, Herodotos, for studying defects over time. The methodology is based on the use of the Coccinelle tool [5, 24, 25, 26, 28] for finding defects, and `diff` [20] for inferring code modifications. Based on this information, Herodotos correlates reported defects across versions and thus discovers the history of each defect. It also allows the user to intervene to specify the false positive reports in order to improve the precision of the results. Finally, Herodotos computes some statistics about defect histories.

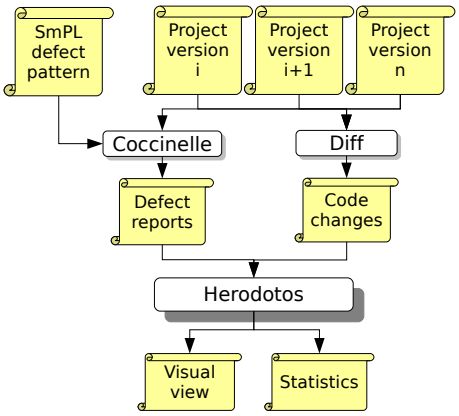

Figure 1: Process overview

The contributions of this paper are as follows:

- We propose a new methodology to study and correlate defects over time.
- We provide a tool, Herodotos, which builds defect histories, generates graphical representations of them and computes defect statistics. For this purpose, it uses defect reports generated by a static code analyzer, in our case Coccinelle, together with information about code modifications generated by `diff`.
- We evaluate the proposed methodology and Herodotos on a representative range of infrastructure software. In doing so, we demonstrate the practicality and usability of our methodology and tools. The selected software projects range from the Linux operating system, to the OpenSSL security library, to the VLC multimedia client.

The remainder of this paper is organized as follows. In Section 2, we present our methodology, including the kinds of defects we are looking for and a step by step description of the approach. In Section 3, we present in detail the Herodotos tool, which is used to correlate defects. In Section 4, we describe our experiments on se-

Defect kinds			CWE
Resource management		Memory	742 401 476
		File	404
Structure	Useless code	No effect code	563
	Insecure code	Coding style	547
	Erroneous code	Bad use of operators	596 682

Table 1: Defect classes

lected software projects and evaluate our methodology based on these experiments. Finally, Section 5 presents some related work and Section 6 concludes.

2 Methodology

Our methodology for obtaining a defect history of a software project is based on the following steps. First, we choose some defect categories that we are interested in. Then, we use a static code analyzer to automatically generate defect reports for these categories. For the files that are found to contain defects, we use `diff` to automatically identify the changes that have occurred from each version to the next one. Finally, we use Herodotos to semi-automatically build defect histories based on the previously computed information.

2.1 Defect classification

To obtain a representative view of how and why defects have been introduced into a software project, it is necessary to have a representative view of these defects. Defects have been studied and classified many times [6, 17, 21]. A widely cited reference is the Common Weakness Enumeration (CWE) [21] where each weakness is explained and detailed with examples. Table 1 describes a range of entries in the CWE, relating to resource management and code structure. We study defects of these classes in our evaluation, presented in Section 4.

2.2 Finding defects

Prior to constructing a defect history, one has to find defects in multiple software versions. In order to have a consistent base of defects, we have chosen to rely on a static analysis tool. For concreteness, we have selected the open-source tool Coccinelle to perform the static analysis, for its availability and flexibility [15, 25, 26, 28]. Coccinelle allows bug finding rules to be specified using a notation that is close to C code, and embeds Python, to allow generating customized output. This facility enables generating defect reports in the notation used by Herodotos.

For each defect class that is of interest, the user creates a bug finding rule for one or more specific kinds of defects. These rules may be inspired by the examples provided in the CWE. They are then applied to each selected version of a software project using the static analysis tool, producing a list of defects. Our approach relies on this list being formatted for use with the Emacs Org mode [23], which creates links to the defect sites in the project source code. Each generated link should contain information related to a defect such as the filename, line and column and possibly some user-defined information. Navigating in this list and following the links allows quickly checking reported defects.

2.3 Computing version to version modifications

When a defect is not corrected by a software developer, it will appear in, and thus be reported for, multiple versions. If defects remain on the same line, with the same surrounding context, they are easy to correlate. But this is unlikely, as unrelated changes may occur elsewhere in the same file. To be able to correlate defects across versions, we thus need to know what other changes have taken place in the same file. Based on the defect reports, a list of the files where a defect may occur is built. For each of these files, the changes from one version to the next are computed by the tool `diff`. They are finally merged for a given project into a single *change* file to have a consolidated set of modifications.

2.4 Herodotos

For each software project, we have at this point a defect report and a change file. Given this information, Herodotos correlates defects across contiguous versions, taking into account code modifications described in the change file. Once the defects are correlated, Herodotos generates a graphical representation of the lifetime of each defect during the period covered by the studied versions. The creation of the file, the introduction of the defect, the deletion of the defect and

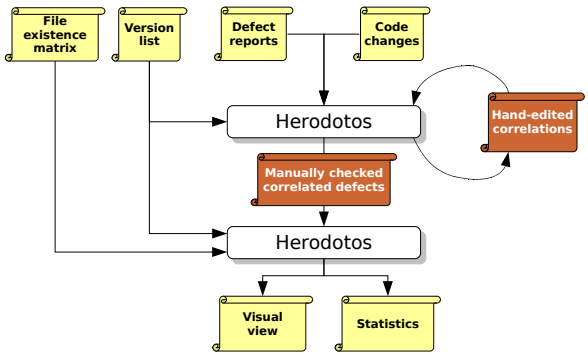

Figure 2: Description of the iterative process with human intervention

the deletion of the file are all illustrated in the graphical representation if they occur during the studied period. Finally, Herodotos computes some statistics about defects in the studied project: (1) the number of affected files, (2) the average number of defects they contain, (3) the number of defects introduced by a new file, (4) the number of defects deleted together with the file they belong to, (5) the minimum, maximum and average lifespan of the defects and (6) the number of defects per subdirectory. Other graphs and statistics may be easily added. We have for instance recently added the generation of a graph that shows the evolution of the number of defects.

3 Herodotos in Detail

Figure 2 illustrates in more detail how Herodotos uses defect reports and information about code changes to generate a defect history, and how the user can interact with the tool. Herodotos works in three main steps: 1) proposing an initial correlation of defects across versions, 2) checking and refining this correlation in response to user-provided hints, and 3) generating graphs and other statistics to describe the defect history of the software. The user can intervene in this process in two ways: i) after step 1, by providing correlation hints, and ii) after step 2, by removing proposed defects that are not actual bugs from the correlated defect list. We outline this process below.

3.1 Correlating defects

Herodotos must correlate identical defects across multiple versions, even when other changes occur in the file. To address this issue, Herodotos uses the GNU `diff` tool to find the differences between the two versions, in order to translate the position of a defect site in one version to its expected position in the next version. For a pair of files, `diff` produces a sequence of *hunks*, which are correlated contiguous sequences of lines that are removed from or added to the first file to produce the second one. `diff` furthermore annotates each hunk with its position, consisting of the starting line number in the first file, the number of lines removed by the hunk, the starting line number in the second file, and the number of lines added by the hunk.[1]

To correlate defects in two successive versions, Herodotos searches for the last hunk whose starting line is at or before the line containing a given defect. If the defect occurs within the lines of code removed by this hunk, Herodotos considers that the defect has been removed. Defects that appear in added lines are considered as new. On the other hand, if the defect occurs after the end of the hunk, then the same code is still present in the file. However this code has been shifted by all the modifications performed by the hunks before it. Herodotos computes a predicted line number for the defect in the next version, by first computing the difference between the number of lines added and removed by the hunk, and then adding this difference to the defect position in the current version. Herodotos then searches for a defect report in the next version involving the same code and at the predicted line. If a defect report is found, the two defect reports are considered to be correlated. If no defect report is found, it typically means that the defect has been fixed by some other adjustment to the file.

This correlation process is illustrated by Figure 3 for an arbitrary version n and its successor. Note that its successor is not necessarily the next revision or release but a user-defined subsequent version, which gives flexibility to our approach. In this example, the lines 10, 20 and 30 are affected by a common defect, *i.e.*, a nonsensical bit-and operation (`&`), and an unrelated change occurs in line 15. The defect on line 30 is fixed between the two versions. We wrote a semantic match for Coccinelle that matches such nonsensical operations. Ap-

[1] We describe here the information provided by the unified mode of `diff` without context, which is the -U0 option. However, the standard mode provides equivalent information.

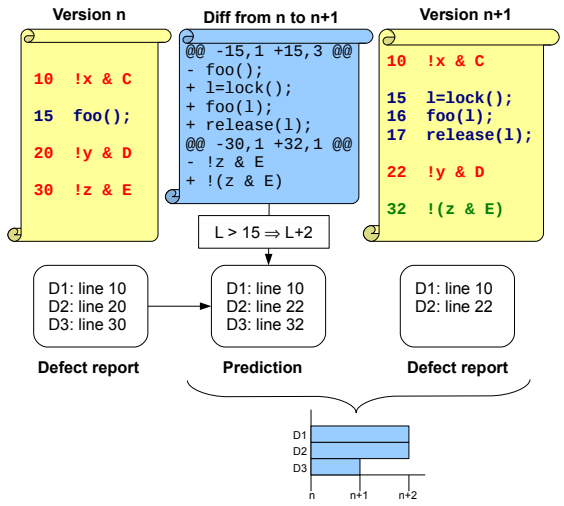

Figure 3: Description of the correlation process

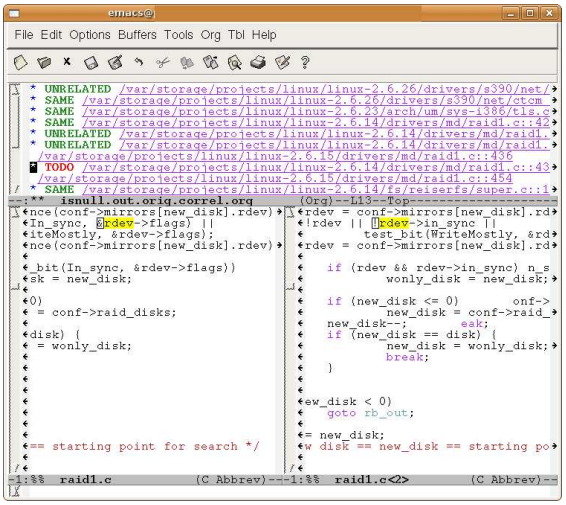

Figure 4: Manually edited file for providing correlation hints

plying it reports the defects $D1$, $D2$ and $D3$, of version n and $D1$ and $D2$ of version $n+1$. Herodotos determines for each report which hunk position must be considered from the information provided by `diff`: for $D1$, no hunk applies; for $D2$, the first hunk applies and lines after the line 15 have now a positive offset of 2; for $D3$, the second hunk applies and the offset is +2. It then computes a prediction of the report in version $n+1$. In this prediction, $D1$ is still at line 10, $D2$ and $D3$ are now respectively at lines 22 and 32. By comparing the prediction with the real report of version $n+1$, Herodotos infers that $D1$ and $D2$ are still present and $D3$ has been fixed as there is no report in that version. If a report was present in that version, *e.g.* parenthesis were not added at line 30 but constant E has changed, Herodotos would have proposed a correlation.

Each software project has its own convention for naming the released versions. The user thus provides Herodotos with an ordered list of version names. This list is used by the correlation process to define the successor version when it looks for matching defects.

3.2 Proposing some corrective information

When there is modification within a line that contains a defect, Herodotos is not able to perform the correlation automatically because the defect is considered to be part of the code removed in some hunk, even though the defect is still present in the file. In this case, Herodotos infers that the defect disappears in some version of the file and another defect appears in the next version.

In this situation, we rely on the user to complete the defect history. To help the user to provide hints as to which defects should be correlated, Herodotos generates a list of possible correlations consisting of all pairs of possibly related defects within a given file. This list is in the Emacs Org mode format with hyper-links to the source code. As illustrated by the screenshot in Figure 4, each pair of possibly related defects is annotated with a state, either `TODO`, `SAME` or `UNRELATED`, indicating respectively a pair to check, an associated pair of defects, or a unrelated pair. In the initial list generated by Herodotos, all pairs have the state `TODO`. For each `TODO` pair, the user can follow the hyperlinks to check the reported defects in a version and the next one. If it appears that both defects in a pair are the same, at possibly different positions, the user changes the state from `TODO` to `SAME`. If the reported defects are unrelated, the user changes the state to `UNRELATED`.

As Herodotos proposes a correlation for each pair of defects within a given file that are possibly related, the number of proposed correlations is potentially quadratic. Each defect, however, can only be correlated with at most one other. Thus, Herodotos filters propositions involving defects that are automatically correlated, and when the user identifies some correlations as valid, Herodotos can automatically eliminate other cor-

relations that are no longer possible. This leads to an iterative process, in which Herodotos takes as input not only a defect report and a set of code changes, but also a partial set of correlations, and produces a new set of possible correlations for the user to validate. A fixed point is reached when the user has identified all proposed possible correlations as `SAME` or `UNRELATED`.

3.3 Checking bug reports

In applying static analysis to a large software project written in a fully-featured language such as C, false positives are essentially inevitable due to limitations in static analysis. Thus, user intervention is required to check the reported bugs. This check could be performed before using Herodotos, but each defect would potentially have to be checked in every version in which it occurs. Thus, Herodotos enables the user to identify false positives after defect correlation but before generation of the final defect history report. In this case, the user has to check only one instance in the lifetime of each defect and accurate results are produced.

As illustrated by the screenshot in Figure 5, each defect history is represented by a headline describing the defect followed by a set of defect positions in each software version. Each of these positions is represented by an Emacs Org mode hyper-link and a state is associated to the headline. A defect state is either `TODO`, `BUG` or `FP`, respectively representing a defect to check, an actual bug or a false positive. All entries initially have the state `TODO`. For each entry, the user can follow the hyperlinks to study the relevant code, and then change the state to `BUG` or `FP`, as appropriate. Only `BUG` defects are considered for rendering in graphical views, and used to compute the statistics.

3.4 Building graphical views of defect histories

Once the actual defects have been identified, Herodotos generates graphical views and computes various statistics. For the final Herodotos execution, the user provides the report containing the checked and correlated defects, the ordered version list, and a matrix indicating the existence of the affected files.

The existence of files will be shown in the detailed graphical view and used to compute the number of defects introduced or removed at the same time as the file they belong to. This can helps in studying the possible conditions that lead to defects. Indeed, popular wisdom

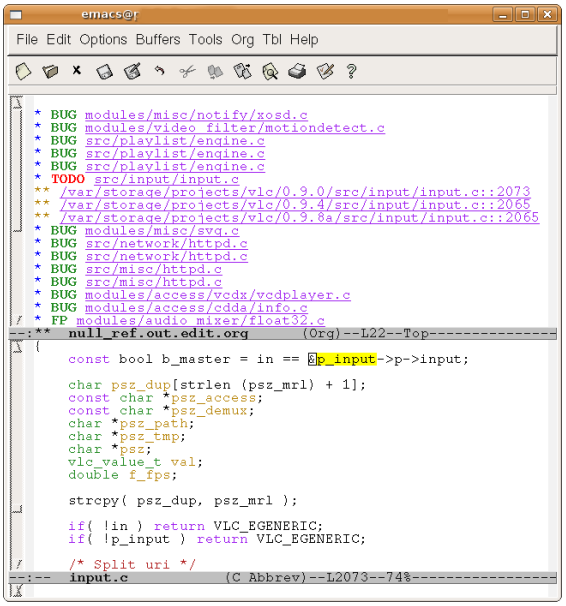

Figure 5: Manually edited file for checking defect false positives

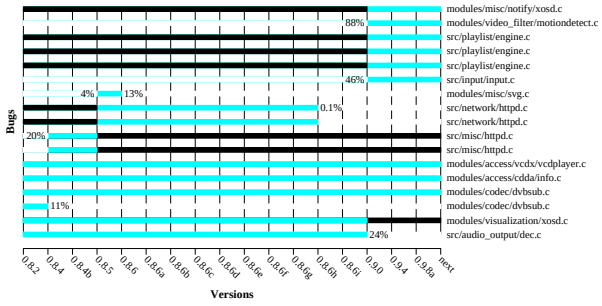

Figure 6: The graph generated for VLC for the NULL reference defect

has it that introducing new code and new features also introduces new defects. The matrix indicating for each file containing a defect whether the file exists in each version, is automatically built from the checked defect reports and the code repositories.

Herodotos creates a graph for each defect kind. As shown in Figure 6, for each defect, there is a light blue/grey bar running from the version in which the defect was introduced to the version in which it disappeared. A black bar indicates the span of versions in which the file does not exist, either because it has not yet been added or because it has been removed. For example, in Figure 6, the first defect was intro-

duced when its containing file was added to the software project, while the second defect was the result of somehow modifying existing code. In this case, the file was substantially rewritten, by more than 80%. For this example, the versions that introduce defects also contain many changes as compared to the corresponding previous versions. In the seventh line, the defect was introduced when adding a missing feature to existing code in version 0.8.5 and it was corrected in version 0.8.6. Finally, the second-to-last defect disappeared because its file was removed.

Herodotos also provides an option to generate compact graphics, to get an overview of the history of all of the considered defects in the software project. In that case, the names of the files containing the defects are omitted and each defect bar is concatenated to its neighbors. Moreover, it generates a summary graphical view. This view shows the evolution of the number of defects and is illustrated Figure 7. In this case, the graph gives the evolution of the number of defects involving comparison of pointers with zero instead of NULL (referred to subsequently as the badzero defect) in four C open-source projects.

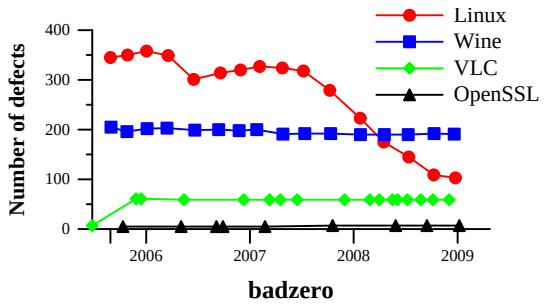

Figure 7: Generated graphical view representing the evolution of badzero for each software project

4 Evaluation

We have applied our methodology on four open source software projects. In our evaluation, we consider both the ease of use of Herodotos and the information that it can give about the defect history of the software projects.

Wine (August 2005 – December 2008)					
20050830	0.9	0.9.5	0.9.10	0.9.16	0.9.21
0.9.26	0.9.30	0.9.36	0.9.41	0.9.47	0.9.54
0.9.60	1.1.1	1.1.6	1.1.11		

VLC (June 2005 – December 2008)						
0.8.2	0.8.4	0.8.4b	0.8.5	0.8.6	0.8.6a	0.8.6b
0.8.6c	0.8.6d	0.8.6e	0.8.6f	0.8.6g	0.8.6h	0.8.6i
0.9.0	0.9.4	0.9.8a				

OpenSSL (July 2005 – September 2008)				
0_9_8a	0_9_8b	0_9_8c	0_9_8d	0_9_8e
0_9_8g	0_9_8h	0_9_8i	0_9_8j	

Table 2: List of versions used

4.1 Selected software

To conduct our evaluation, we have selected four software projects with different profiles: Linux [18], Wine [31], VLC [29] and OpenSSL [22]. These have been selected to cover aspects ranging from a full operating system (Linux), to an OS user interface (Wine), to a user-level multimedia application component (VLC). OpenSSL was selected to compare these aspects against security concerns.

Of these software projects, Linux has the most stable release model, with a new release occurring roughly every three months. Thus, we have selected Linux as the reference project. For Linux, we have studied every release from v2.6.13 (August 2005) to v2.6.28 (December 2008), inclusive. For the other projects, we have selected releases occurring at about the same time as the Linux releases. The releases for the other projects are given in Table 2. Two versions are missing for OpenSSL, because the corresponding archives are corrupted, even in the mirror servers.

Figure 8 shows the number of lines of C code in each software project across the different versions. Linux is the largest software project, with between 4 million and 6 million lines of C code in the considered time period. Wine is the next largest, with between 1 and 1.5 million, and OpenSSL and VLC are the smallest, with between 200,000 and 330,000 lines of C code. Figure 9 shows the increase in the number of lines of C code in each software project across the different versions, as compared to the first considered version. Within the considered time period, Linux, Wine, and VLC have increased in size by around 50%, while OpenSSL has only increased in size by around 15%. For VLC, the code size remained essentially the same for a long period, and

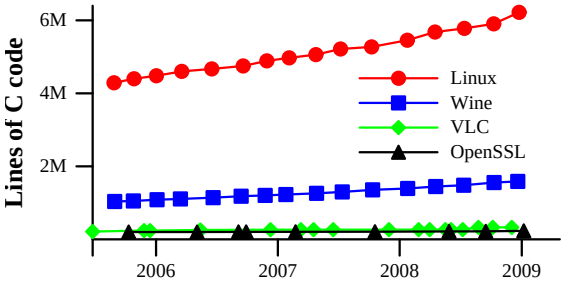

Figure 8: Code sizes of the software versions released between 6/25/2005 and 02/13/2009

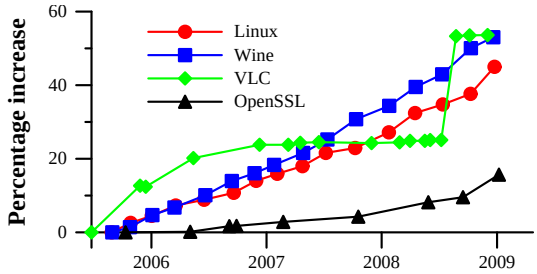

Figure 9: Increase in code size as compared to the first version released on or after 6/25/2005

then spiked, while for the other software projects, the increase in code size has been more linear.

4.2 Selected defect kinds

We have selected a range of defect kinds in the classes shown in Table 1. To allow comparison between the projects, the selected kinds are project agnostic and could be applied to any C project, except for Linux where `malloc`/`free` has been changed to `kmalloc`/`kfree`. These are as follows: 1) A file descriptor is acquired, but not released (`open`), 2) memory is allocated, but not freed (`malloc`), 3) dereference of a NULL pointer (`isnull`), 4) dereference of a value and subsequent checking that the value is NULL (`null_ref`), 5) checking that a pointer is NULL, when it is already known that it is not NULL (`notnull`), 6) assigning a constant to a variable, but never using the variable's value (`unused`), 7) comparing a pointer to 0, rather than NULL (`badzero`), 8) misusing boolean and bit operations (`notand`), and 9) checking that an unsigned value is less than 0 (`unsigned`). Some of these defects can cause a crash or memory leak (*e.g.*, `isnull` and `malloc`). Others do not intrinsically cause incorrect runtime behavior, but can be a symptom of some other bug. For example, in our experiments, we have found code where a second redundant NULL test (`notnull`) should have been a test on a different value. Others simply make the code more difficult to understand. For example, comparing the result of a pointer typed expression to 0 rather than NULL (`badzero`) suggests that the value of the expression is an integer.

4.3 Experiments

We have performed our experiments on a HP ProLiant server with two 3GHz quad-core Xeon processors and 16 GB memory. The combined size of the software projects is 8GB. Running Coccinelle on this code base for the defect kinds described in Section 4.2 took a couple of days, with most of the time spent on Linux. Once the results were generated by Coccinelle, we used Herodotos to correlate the 22,135 reported defects in about five minutes. We then checked the reported defects for false positives, and used Herodotos to build the graphical representations and compute the corresponding statistics. This last execution of Herodotos only takes a few seconds. In the next section, we present a synthesis of these statistics from several points of view: in terms of the software projects, the defect classes, and the evolution of the defects through the project history.

4.4 Usability

For Linux, Wine, VLC and OpenSSL and the given defect-finding rules, Coccinelle reports over 22,000 defects for a period of over 3 years. As shown by Table 3, Herodotos automatically infers more than 99% of the correlations between them. 19,257 correlations are performed automatically, while only 220 are initially proposed to the user for review. Based on this set, and using the iterative process, the user needs to explicitly annotate only 146 correlations. 108 of these are annotated as `SAME` while the 38 others are annotated as `UNRELATED`.

Through these experiments, Herodotos has shown its usability and effectiveness in correlating reports of the same defect across multiple versions. The current computation of statistics was easy to develop and integrating new ones should not be difficult. Finally, the graphical representations can help developers see at a glance what happens in their software. The summary view gives for instance an idea of the project improvement over time,

	Reported defects	Automatically correlated defects	Initially proposed correlations	User provided correlations		Correlated defects
Linux	16,532	14,081	142	118	0.84%	1,931
Wine	4,140	3,746	75	25	0.67%	381
VLC	1,313	1,152	2	2	0.17%	158
OpenSSL	331	278	1	1	0.36%	52
Total	22,316	19,257	220	146	0.76%	2,522

Table 3: Correlation effectiveness

while the detailed view could point out to the user some files to inspect and gives some feedback about the manual phases, the correlation hints and the false positive inspection.

4.5 Results

Tables 4 and 5 and Figures 7, 10 and 11 summarize the results of the defect-finding process as well as the statistics produced by Herodotos. Table 4 presents the number of defects found by Coccinelle in the various defect classes, including false positives, and the number that we have confirmed by a manual inspection. Table 5 describes the ratio of found defects to the number of code sites that are relevant to the defect kinds. For example, for the first two resource defects, we count the number of calls to `open` or `malloc` (or its Linux-specific counterpart `kmalloc`), respectively. Similarly, for the defect of double checking a pointer for `NULL`, we count the number of `NULL` tests. For `notand` and `badzero`, we respectively count the number of bit-and operations and the number of comparisons with zero. There is no entry for `unused` in this table because it was not clear what kind of site would be relevant. Figure 10 shows the number of occurrences of each kind of defect at any point in the considered time period for each software project, the average lifetime of each such defect, and the number of such defects that have been introduced when adding a new file to the software project, and removed when deleting a file. Finally, Figures 7 and 11 show the evolution in the number of defects throughout the studied period.

Per software project As shown in Table 4, Linux, as the largest software project we have studied, tends to have the highest number of defects and has at least one defect for every defect-finding rule. `open` is a special case; the rate of defects per possible defect site is 3%, but there are only 3 defects. As the programs involved are small with a secondary purpose, the developers may intentionally rely on the automatic mechanisms provided by the OS to close file descriptors. The critical aspect of the OS has also an impact on the overall quality. Not counting the `open` special case, Linux has average ratio of 0.36%, which is better than Wine, and the lowest maximum ratio of defects (0.92%) except for OpenSSL (0.42%). However, OpenSSL is 24 times smaller than Linux and is dedicated to security. Linux has nevertheless the maximum ratio of confirmed defects to possible defect sites for a third of the defect kinds.

Wine has many similarities with Linux. As shown in Figure 9, they have been growing at a similar rate. In a number of cases, Linux and Wine have a similar percentage of defects per possible defect site (Table 5). Wine has also at least one occurrence of each defect kind, except `open`, unlike OpenSSL and VLC.

VLC does not distinguish itself from the others with an extreme value. Its number of defects is relatively low but three times higher than that of OpenSSL, which is comparable in terms of the code size. It also has around three times the number of defects per relevant site as compared to OpenSSL, both in terms of the average and the maximum (Table 5).

Finally, OpenSSL confirms its position as stable security software. Indeed, as shown in Table 4 and 5, it has the lowest absolute number of confirmed defects and its maximum ratio of defects is also one order lower compared to the three other projects. The developers of OpenSSL are also very conservative, which leads to long defect lifetimes, as shown in Figure 10(b).

Per defect category Figure 10 presents the statistics computed by Herodotos: the number of defects, the average lifespan, the number of defects introduced and removed at the same time as the file they belong to. The defect kinds are presented in the same order as in Tables 4 and 5.

Defect kinds		Software projects							
		Linux		Wine		VLC		OpenSSL	
		C. /	R.	C. /	R.	C. /	R.	C. /	R.
Resource	File descriptor not released	3 /	4	0 /	0	0 /	0	0 /	1
	Memory not released	42 /	44	2 /	2	2 /	2	0 /	0
	Dereference after NULL	42 /	92	1 /	4	3 /	6	3 /	4
	Dereference before checking NULL	276 /	309	19 /	27	17 /	20	4 /	7
Useless code	Double check a pointer with NULL	48 /	69	30 /	30	4 /	4	11 /	13
	Assign a constant to an unused variable	267 /	286	36 /	46	8 /	9	16 /	18
Insecure code	Compare with zero instead of NULL	851 /	862	7 /	7	115 /	115	248 /	255
Erroneous code	Wrong use of ! with &	72 /	76	16 /	16	2 /	2	0 /	0
	Check if an unsigned value is less than 0	188 /	189	1 /	1	0 /	0	2 /	2
	Total	1,789 /	1,931	356 /	381	151 /	158	43 /	52

Table 4: Reported and confirmed defects by defect categories

Defect kinds	Software projects				Min	Avg	Max
	Linux	Wine	VLC	OpenSSL			
File descriptor not released	3.09%	0.00%	0.00%	0.00%	0.00%	0.77%	3.09%
Memory not released	0.53%	2.30%	0.16%	0.00%	0.00%	0.75%	2.30%
Dereference after NULL	0.01%	0.001%	0.01%	0.02%	0.001%	0.01%	0.02%
Dereference before checking NULL	0.10%	0.01%	0.03%	0.02%	0.01%	0.04%	0.10%
Double check a pointer with NULL	0.05%	0.10%	0.05%	0.09%	0.05%	0.07%	0.10%
Compare with zero instead of NULL	0.82%	0.80%	1.49%	0.06%	0.06%	0.79%	1.49%
Wrong use of ! with &	0.08%	0.12%	0.10%	0.00%	0.00%	0.07%	0.12%
Check if an unsigned value is less than 0	0.92%	0.09%	0.00%	0.42%	0.00%	0.36%	0.92%
Minimum	0.02%	0.001%	0.00%	0.00%			
Average	0.70%	0.49%	0.26%	0.09%			
Maximum	3.09%	2.30%	1.49%	0.42%			

Table 5: Ratio between confirmed defects and possible sites for this defect

Despite the fact that many static and dynamic analysis tools have considered defects related to the release of allocated memory, Figure 10(a) shows that many such errors still exist, particularly in Linux code, where user-space tools such as Valgrind can not be used to detect memory related defects. In the case of Wine, this issue has the highest ratio of checked defects to potential defect sites. These defects, however, tend to have a shorter lifespan, as shown in Figure 10(b), particularly for `malloc` in Wine or `isnull` for the other projects.

Other defects that have a longer lifespan in certain software projects include deferencing of a NULL value (`isnull`, in Wine), redundant NULL tests (`notnull`, in VLC), comparing a pointer to zero (`badzero`, in Wine), and the OpenSSL defects in general. Of these, only the first can lead to a program crash. Except for Linux, in which such defects have a relatively short lifespan, the number of such defects is small. The non critical aspect of the three other kinds of defects could explain why they have a longer average lifespan.

The defect of comparing a pointer to zero is common in newly added files (Figure 10(c)) and has a long lifespan (Figure 10(b)), with the defect typically often being either still present or removed only when the file disappears (Figure 10(d)). Finally, testing whether and unsigned integer is less than zero is a common error, occurring at almost 1% of such comparisons with 0 in Linux.

Evolution through the project history Figure 11 shows the summary graphs generated by Herodotos. For each project and each kind of defect, a line gives the evolution of the defects. The lines related to `badzero` are plotted in a separate graph to improve the readability (Figure 7).

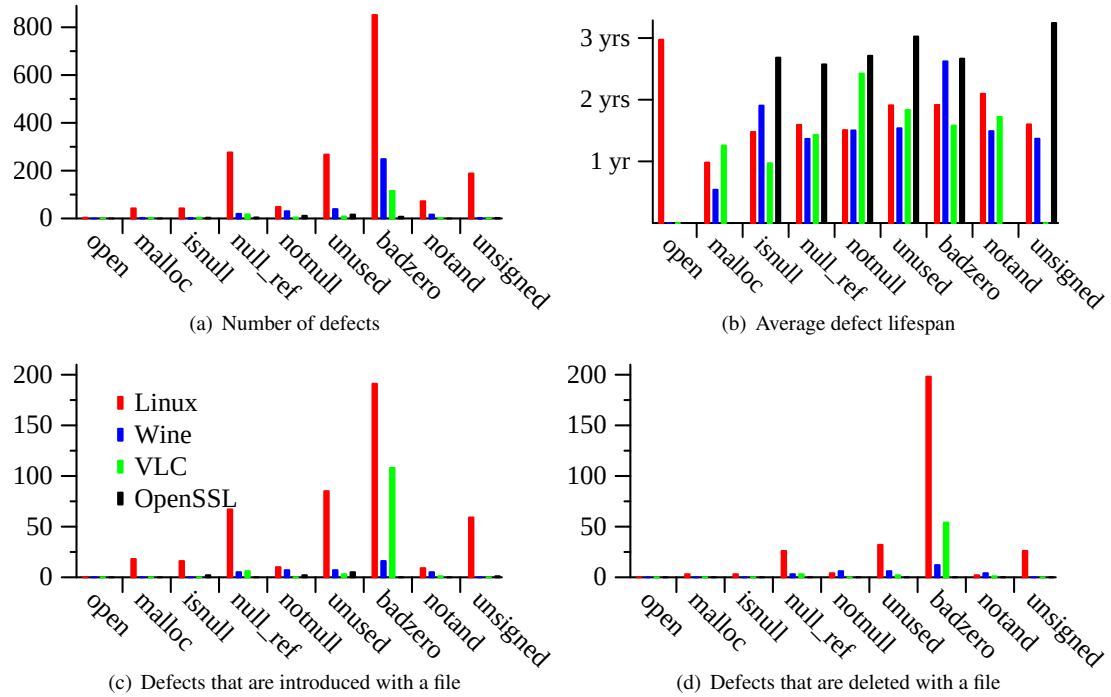

Figure 10: Generated statistics for each defect kind and each software project

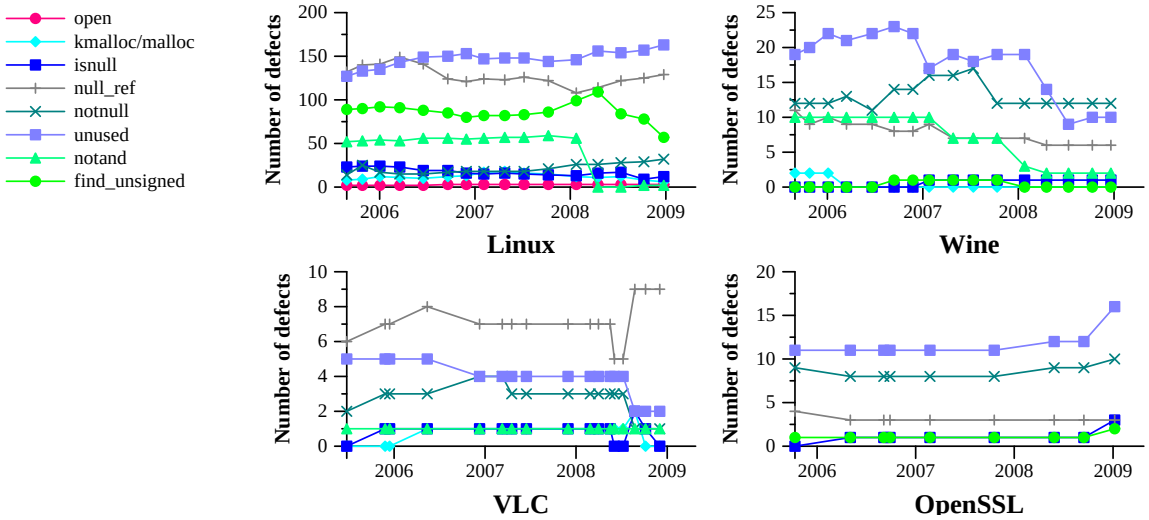

Figure 11: Generated graphs representing each defect evolution for each software project

We observe that the reported defects about misuse of boolean and bit operators (`notand`) and comparison of unsigned with zero have declined dramatically since versions 2.6.24 and 2.6.25, respectively. As we have been using Coccinelle for finding and fixing bugs in the Linux kernel since version 2.6.24, it could explain in part the trend we observe on the later studied versions. However, we still find some defects and it would be now interesting to learn why they have not been fixed. Has a patch already been submitted? If so, why it is not incor-

porated? Has it been lost or rejected? Finally, we do not observe a similar trend for the other kinds of defects.

Wine is the second largest project studied in this paper. It has a lot of unused variables, which have been cleaned up in 2008. We also note that misuse of boolean and bit operators (`notand`) has declined significantly. This work was begun in 2007.

VLC, which is less critical than the other selected projects, distinguishes itself by a poor coding style with a relative high rate of comparisons of pointers to 0 and unused variables. However, unused variables have been recently cleaned up. We also note that after a decrease in NULL dereferences (`isnull` and `null_ref`) in mid 2008, the number of this kind of defect has increased in the last three studied releases. Others defect kinds are stable over time.

Defects in OpenSSL are stable which suggests that little effort has been made to fix them. The number of unused variable defects has even recently risen. However, OpenSSL is also the project with the lowest rate of growth, around 15% (Figure 9), which suggests that its development is not very active even though it is widely used.

4.6 Threats to validity

In this section, we investigate possible threats to the validity of this study. Although some of them have been already mentioned elsewhere in this paper, they are summarized with the others here.

Static code analysis The use of a static code analysis rather than runtime testing means we may have some false positives in the defect reports. Moreover, the defects we are looking for must match the manually written semantic match. In some cases, such as `malloc`, the semantic match is fairly restrictive, to prevent false positives. This strategy may nevertheless lead to false negatives. Furthermore, we have primarily relied on our own expertise to distinguish true defects from false positives; although for each bug type, we have submitted and had accepted a number of patches to the Linux kernel or noted patches related to the bug type that have been submitted by others.

Renaming We currently have a fairly restrictive definition of a *identical* bug across versions. As our methodology does not deal with variable, file and directory renaming, it leads to an artificially high number of defects and rate of defect turnover. To eliminate this limitation, we need to agree on a generalized definition of identical bug. Nevertheless, the most appropriate definition might be specific to the goal the study, the defect kind, and the scale, *i.e.*, variable, file or directory renaming.. Defining what is an identical/unique bug across versions thus remains an open question.

Representativeness We have chosen the four aforementioned projects for their heterogeneity, with the goal of comparing a wide variety of projects. Nevertheless, due to time constraints and the large volume of code and defects involved, it has been possible to consider only one project in each category. A chosen project may not be representative of its category.

The nine semantics matches used are project agnostic. Generic bugs may not be representative of all of a project's bugs. Using mechanisms to infer and then fix software-specific bugs [15, 16] can provide semantic matches to study software-specific bug histories.

5 Related work

Previous work has considered either static code analysis to find defects [2, 6, 7, 8, 30] or defect history based on bug trackers [10, 19], but little has been done to build tools to track defects based on code modifications. Chou *et al.* [6] have made a detailed study of the history of 12 kinds of bugs in Linux code up to 2001. Their study involves the automatic propagation of reported defects across successive versions but no explanation was given as to how this was done and no tool was released. More recently, Li *et al.* [17] have conducted an empirical study on open source software. However, no tool to automatically or semi-automatically build defect histories was mentioned and the bugs considered came from a bug tracker system.

More recent work has also considered the automatic correlation of defects, or warnings. Spacco *et al.* [27] pursue the same goal but use the warning message as the warning identifier with which to compute a hashcode. In their definition, identical defects have the same hashcode and thus identify a unique defect history. In contrast, our approach is based on the exact position in the file. Kim and Ernst [12] rely on the log messages provided by the developers in the source code management (SCM) system to identify defects and prioritize warnings. This approach assumes that developers consistently use a set of keywords to characterize

each commit. Moreover, every line modified by a commit has the same status as the others, bug-related or non-bug. Extraneous modifications, such as removal of trailing spaces, may thus mark the entire line as a bug in a commit designated as a bug fix by its changelog. Whatever is the status of the committed lines, it is back ported to the previous revisions, thus propagating erroneous status on some lines. Finally, the authors do not explained how the line status is propagated beyond the first previous revision, when other commits should be taken into account and code has changed. Boogerd and Moonen [4] use a history collecting mechanism based on the two previous ones. It thus suffers from the same strong dependency on the SCM system. Their aim is to study the use of coding standards and their impact over time on software projects.

Static code analysis has been used for finding defects in upcoming releases or recent ones, but without consideration of a long period of time to build a defect history. Defect history has, however, been studied by coupling an SCM system, generally CVS, with a bug tracking system. DynaMine [19] applies data mining techniques to the data collected by an SCM system to find frequent application-specific coding patterns that can be subsequently used to check for bugs. ROSE [32] uses the same technique to suggest modification sites during software evolution. Finally, iBUGS [9] explores information contained in an SCM system to infer bugs and associated tests in order to build a benchmark for defect searching tools. However none of these approaches uses a static code analyzer.

In our experiments, we have used the standard GNU diff [20]. However, using the Patience diff algorithm [1, 3] could have lead to more human readable difference files and maybe a better correlation in our case. Unfortunately, the tool implementing this algorithm needs some modifications to be integrated with our approach. Due to time constraints, these modifications have not yet been carried out.

6 Conclusion

In this paper, we have presented the tool Herodotos which tracks defects through software releases, builds a graphical representation of the history of these defects and computes some statistics. This process leverages existing tools to infer code modifications and defect positions. It then automatically builds the history of each defect. To overcome the inherent imprecision of the tools on which it relies, Herodotos provides the user the means to intervene in the process. The user can provide information to improve the correlation between versions and eliminate false positives reported by the static code analyzer. Herodotos assists the user in providing this information by proposing some possible correlations based on heuristics.

In future work, we are considering how to exploit more information from source code management systems to improve the automatic correlation process. For instance, Herodotos is currently not able to correlate defects when a file is renamed or moved to another directory. The git SCM system [11] tracks content, and not just file names, and from this information is able to infer file or directory renaming. Using this information will help Herodotos to automatically infer more correlations. Tightly coupling Herodotos with a SCM system, and further extending it with an interface to a bug tracking system, will make it possible to determine why and how bugs have been found and fixed. Finally, the Coccinelle static code analyzer allows the user to define software-specific defect patterns [16]. We plan to exploit this feature to study software-specific bugs and look for new defect categories from a software-specific point of view.

In the evaluation of Herodotos, we have shown on the four studied projects that the number of defects tends to be either stable or raising. In the case of Linux, the result of using Coccinelle, for bug detection and fixing, has shown some visible effects. A more automatic use of both Coccinelle and Herodotos will thus be desirable. It will aid software developers to fix defects and understanding the overall improvement.

Availability Herodotos is available at `http://www.diku.dk/~npalix/herodotos/` with a page presenting the results.

References

[1] Alfedenzo. Patience diff, a brief summary. `http://alfedenzo.livejournal.com/170301.html`, fev 2008.

[2] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *The first ACM SIGOPS EuroSys*

conference (EuroSys 2006), pages 73–85, Leuven, Belgium, April 2006.

[3] Sergei Bespamyatnikh and Michael Segal. Enumerating longest increasing subsequences and patience sorting. *Information Processing Letters*, 76(1-2):7–13, November 2000.

[4] Cathal Boogerd and Leon Moonen. Assessing the value of coding standards: An empirical study. In *IEEE International Conference on Software Maintenance, 2008. ICSM 2008*, pages 277–286, Beijing, China, September 2008.

[5] Julien Brunel, Damien Doligez, René Rydhof Hansen, Julia Lawall, and Gilles Muller. A foundation for flow-based program matching using temporal logic and model checking. In *The 36th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 114–126, Savannah, GA, USA, January 2009.

[6] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, pages 73–88, Banff, Canada, October 2001.

[7] Patrick Cousot, Radhia Cousot, Jérôme Feret, Antoine Miné, Laurent Mauborgne, David Monniaux, and Xavier Rival. Varieties of static analyzers: A comparison with ASTRÉE. In *First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering*, pages 3–20, Shanghai, China, June 2007.

[8] Static source code analysis, static analysis, software quality tools by Coverity Inc. http://www.coverity.com/, 2008.

[9] Valentin Dallmeier and Thomas Zimmermann. Extraction of bug localization benchmarks from history. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 433–436, Atlanta, GA, USA, November 2007.

[10] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 23–32, Amsterdam, The Netherlands, September 2003.

[11] Git: The fast version control system. http://git-scm.com/.

[12] Sunghun Kim and Michael D. Ernst. Which warnings should I fix first? In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 45–54, Dubrovnik, Croatia, September 2007.

[13] Greg Kroah-Hartman, Jonathan Corbet, and Amanda McPherson. Linux Kernel Development: How Fast is it Going, Who is Doing It, What They are Doing, and Who is Sponsoring It. In *OLS'06: The 2006 Ottawa Linux Symposium*, pages 239–244, 2006.

[14] Greg Kroah-Hartman, Jonathan Corbet, and Amanda McPherson. Linux Kernel Development: How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It. http://www.linuxfoundation.org/publications/linuxkerneldevelopment.php, April 2008.

[15] Julia Lawall, Gilles Muller, and Nicolas Palix. Enforcing the use of API functions in Linux code. In *8th Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS '09)*, pages 7–11, Charlottesville, VA, USA, March 2009.

[16] Julia L. Lawall, Julien Brunel, René Rydhof Hansen, Henrik Stuart, Gilles Muller, and Nicolas Palix. WYSIWIB: A declarative approach to finding protocols and bugs in Linux code. In *The 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, (DSN 2009)*, pages 43–52, Estoril, Portugal, June 2009.

[17] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 25–33, San Jose, CA, 2006.

[18] Linux kernel. http://kernel.org/.

[19] Benjamin Livshits and Thomas Zimmermann. DynaMine: finding common error patterns by mining software revision histories. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 296–305, Lisbon, Portugal, September 2005.

[20] David MacKenzie, Paul Eggert, and Richard Stallman. *Comparing and Merging Files With Gnu Diff and Patch*. Network Theory Ltd, January 2003.

[21] Mitre. Common Weakness Enumeration. http://cwe.mitre.org/.

[22] OpenSSL. http://www.openssl.org/.

[23] Org-mode homepage. http://orgmode.org/.

[24] Yoann Padioleau, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. Semantic patches for documenting and automating collateral evolutions in Linux device drivers. In *PLOS 2006: Linguistic Support for Modern Operating Systems*, pages 55–60, San Jose, CA, October 2006.

[25] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Semantic patches for collateral evolutions in device drivers. In *Linux Symposium*, Ottawa, Canada, June 2007.

[26] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in Linux device drivers. In *EuroSys 2008*, pages 247–260, Glasgow, Scotland, March 2008.

[27] Jaime Spacco, David Hovemeyer, and William Pugh. Tracking defect warnings across versions. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 133–136, Shanghai, China, May 2006.

[28] Henrik Stuart. Hunting bugs with Coccinelle. Master's thesis, University of Copenhagen, Copenhagen, Denmark, August 2008.

[29] VLC media player. http://www.videolan.org/vlc/.

[30] David Wheeler. Flawfinder home page. Web page: http://www.dwheeler.com/flawfinder/, October 2006.

[31] Wine Is Not a Emulator. http://www.winehq.org/.

[32] Thomas Zimmermann, Andreas Zeller, Peter Weissgerber, and Stephan Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.

Centre de recherche INRIA Paris – Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399