Tracking Code Patterns over Multiple Software Versions with Herodotos

Nicolas Palix, Julia Lawall, Gilles Muller

Tracking Code Patterns over Multiple Software Versions with Herodotos

Nicolas Palix
DIKU
University of Copenhagen
Denmark
npalix@diku.dk

Julia Lawall
DIKU, University of Copenhagen
INRIA-Regal
Denmark/France
julia@diku.dk

Gilles Muller
INRIA-Regal
LIP6
France
Gilles.Muller@inria.fr

ABSTRACT

An important element of understanding a software code base is to identify the repetitive patterns of code it contains and how these evolve over time. Some patterns are useful to the software, and may be modularized. Others are detrimental to the software, such as patterns that represent defects. In this case, it is useful to study the occurrences of such patterns, to identify properties such as when and why they are introduced, how long they persist, and the reasons why they are corrected.

To enable studying pattern occurrences over time, we propose a tool, Herodotos, that semi-automatically tracks pattern occurrences over multiple versions of a software project, independent of other changes in the source files. Guided by a user-provided configuration file, Herodotos builds various graphs showing the evolution of the pattern occurrences and computes some statistics. 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 track several kinds of defects that have been found by pattern matching. This tracking is done automatically in 99% of the occurrences. The results allow us to compare the evolution of the selected projects and defect kinds over time.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*Process metrics, Product metrics*; D.3.2 [**Programming Languages**]: Language Classification—*Specialized application languages, HCL, SmPL*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Patterns*

General Terms

Measurement, Languages, Reliability

Keywords

History of pattern occurrences, bug tracking, Herodotos

AOSD'10 March 15–19, Rennes and St. Malo, France

1. INTRODUCTION

Patterns have been found to be useful in finding various types of defects or "bad smells" in software source code. Tools such as Coverity [9, 12], Flawfinder [35], and Coccinelle [22] use pattern-based techniques to find defects such as dereferences of NULL pointers, checks whether an unsigned value is less than zero, and memory leaks. These tools have been applied to widely used infrastructure software projects such as the Linux operating system, where safety and security are critical, to find significant bugs. Nevertheless, these tools do not fully realize the potential of patterns for understanding the robustness of a software system, because they consider only one version of the software at a time. Studying occurrences of defects over time can provide information about software quality by identifying long-running or frequently occurring trouble spots and aid in understanding the software development process by clarifying the origins and evolution of certain kinds of defective code patterns.

Tracking pattern occurrences across the history of a large software project is, however, a daunting task. First, it is necessary to identify the relevant code fragments. Second, these code fragments must be correlated across software versions, even in the presence of changes in the source code that may affect the same file, the same function, or even the same line as a pattern occurrence. Finally, it must be possible to manage the collected information, to easily mine it for specific facts and trends.

In this paper, we propose a language-independent approach, illustrated in Figure 1, for mining and tracking code pattern occurrences across a software history. Our approach includes a tool, Herodotos, which relies on the Emacs org mode [30] as an input format for reporting pattern occurrences, and `diff` [26] for inferring code modifications across versions. Based on this information, Herodotos correlates reported pattern occurrences across multiple versions of a software project and thus tracks each pattern occurrence to discover its history. It also allows the user to intervene to identify the reports that are false positives in order to improve the precision of the results. Finally, Herodotos provides a domain-specific configuration language for manipulating the collected information, enabling the user to study the history of the occurrences of a pattern in a variety of output formats.

In our experiments, we have instantiated Herodotos with the Coccinelle pattern matching tool [6, 31], which allows searching for user-specified code patterns in C code. We have evaluated Herodotos on patterns representing software defects on a representative range of open-source infrastructure software: the Linux operating system kernel, the Wine

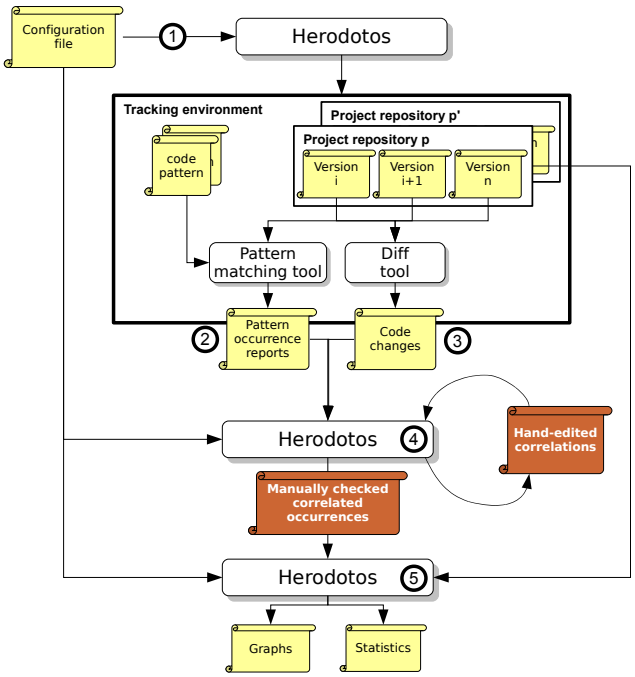


Figure 1: Overview of Herodotos

OS API, the OpenSSL security library, and the VLC multimedia client. This kind of software underlies most of the computing in a modern desktop environment, and can be security-sensitive. Thus, it is critical not only to find code defects, but also to understand their lifecycle, to help develop priorities for the ongoing software development process.

The contributions of this paper are as follows:

- We propose an approach and a tool, Herodotos, for tracking code pattern occurrences over multiple software versions. This tracking reconstructs the history of the occurrences in a software project.
- We propose the Herodotos Configuration Language (HCL) to help the user manage the process of searching for and visualizing pattern occurrences. The user writes an HCL file describing the experimental environment and the information of interest. Guided by this file, Herodotos then generates a variety of graphs and statistics.
- We formalize the strategy used by Herodotos for correlating code pattern occurrences over multiple versions of software, even in the presence of other changes in the file. This strategy allows for user intervention, but in practice we have found that such intervention is rarely needed. For the patterns and projects considered in Section 4, fewer than 1% of the correlations must be considered by hand.
- We show that the information collected by Herodotos can give insights into the robustness and stability of software projects. In particular, we find that large projects such as Linux and Wine tend to have more defect kinds than smaller ones such as VLC or OpenSSL, and that the developers of the OpenSSL security library are so conservative that the number of defects is very stable as compared to other projects.

The rest of this paper is organized as follows. Section 2 describes the steps of our approach. Section 3 presents the two key points of Herodotos: the configuration language and the automatic correlation of occurrences. Section 4 describes our experiments on selected software projects and evaluates our approach based on these experiments. Finally, Section 5 presents related work and Section 6 concludes.

2. APPROACH

Our approach for obtaining a pattern occurrence history in a software project is based on the following steps, illustrated in Figure 1: 1) set up a tracking environment, 2) identify the pattern occurrences in the software, 3) identify the changes that have occurred from one version to the next, 4) correlate pattern occurrences across multiple versions, modulo the identified changes, and 5) generate graphs and statistics representing various aspects of the history of the pattern occurrences. This process is guided by a configuration file that describes the patterns of interest, the software project versions, and the desired kinds of graphs and statistics.

Setting up a tracking environment.

The user must initially choose a pattern matching tool, create the code patterns of interest according to the notation of the chosen tool, and collect the software versions to study, grouped by project. He then declares the means of invoking the pattern matching tool and the location of the patterns and the projects in the configuration file. Finally, he invokes Herodotos with this configuration file to create the tracking environment.

Finding pattern occurrences.

Although Herodotos is independent of the pattern matching tool, it must still be able to interpret the output of this tool, to find the file names and the positions of the pattern occurrences. Herodotos expects this information to be presented using the hyperlink notation of the Emacs "org mode" [30], which makes it possible to generate links from the pattern occurrence reports to the project source code, for easy validation of the collected information.

Based on the information provided in the configuration file, Herodotos applies each of the pattern matching rules listed in the configuration file to every version of each software project and collects the results. This step can be time-consuming, and thus it can be performed offline and the results cached for later processing.

Computing version modifications.

When a pattern occurrence is not removed by a software developer, it will appear in, and thus be reported for, multiple versions. If the pattern occurrences 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 pattern occurrences across versions, we thus need to know what other changes have taken place in the same file. For each file for which there is at least one pattern report in successive versions, Herodotos uses the GNU `diff` tool to compute the set of changes between these two versions.

Correlating and validating occurrences.

Once the pattern occurrences and the version modifications in the affected files have been identified, Herodotos uses this information to correlate the pattern occurrences across the versions of each project. The result is a report containing one entry for each correlated pattern occurrence. Each entry consists of a set of links to the position of the given pattern occurrence in each version of the file in which it occurs. In some cases, Herodotos is not able to perform the correlation automatically. In such cases, it proposes a set of possible correlations to the user, who can accept or reject them.

Pattern matching tools that are based on static analysis, without information about the code paths that are actually executed at run time, are subject to false positives, *i.e.*, code fragments that match the provided pattern, but for some reason are not relevant to the property of interest. Herodotos thus enables the user to intervene at this point, to designate reports as false positives. Such reports are discarded in subsequent processing. By integrating this step into Herodotos, the user only has to check one instance of each pattern occurrence, rather than every instance in every version in which the pattern occurs.

Generating graphs and statistics.

Based on the validated occurrences and various information about the software projects such as the number of lines of code in each file/project, and the lifetime of each file, Herodotos can generate a variety of graphs and statistics, guided by the configuration file. Information is presented at the level of a specific pattern occurrence or as a summary of the properties of the occurrences of a given pattern across multiple projects. The number of pattern occurrences can be presented directly, or it can be correlated with other information, such as the size of each file, the number of changes in each file, and the point of creation and deletion of each file. These graphs and statistics can help understand whether pattern occurrences typically are introduced when a file first appears, are removed when a file is deleted, or are introduced or removed for other reasons. They also show trends in pattern occurrences across a project's history, and permit comparisons between projects.

3. HERODOTOS IN DETAIL

The key technical contributions in the design of Herodotos are the configuration language, which exposes Herodotos' features to the user, and the strategy for correlating pattern occurrences across multiple versions.

3.1 Herodotos Configuration Language (HCL)

When studying pattern occurrences in one or more large software projects, the amount of information to manipulate can quickly becoming overwhelming. To assist the user in managing this information, Herodotos provides HCL, defined in Figure 2. HCL allows the user to specify how to set up the tracking environment and how the history derived from the collected information should be presented.

Concretely, an HCL configuration file specifies four main kinds of information: general configuration parameters, the set of projects, the set of code patterns, and the desired set of graphs and statistics. Each kind of information is described by a set of attributes, *i.e.*, a key/value pair, as defined by the various "*attr*" rules of the grammar. Figure 3 gives an excerpt of the HCL file used for the study presented in Section 4.

```
config     ::= globattr* project* pattern* graph*
globattr   ::= prefix = string | projects = string
             | patterns = string | results = string
             | findcmd = string | flags = string
project    ::= project id {prjattr*}
pattern    ::= pattern id {patattr*}
graph      ::= graph filepath {grphattr* curve*}
prjattr    ::= flags = string | dir = string
             | styleattr
             | versions = {((string,date,int))*}
patattr    ::= flags = string | file = string
             | styleattr
curve      ::= curve project id ({layoutattr*})?
             | curve pattern id ({layoutattr*})?
             | curve project id pattern id {layoutattr*}
grphattr   ::= xaxis = xtype | yaxis = ytype
             | xlegend = string | ylegend = string
             | pattern = id | project = id
             | layoutattr
xtype      ::= version | date
ytype      ::= occurrences | sum | density | cumul
             | size | sizepct | birth | death
             | birthfile | deathfile | avglifespan
layoutattr ::= nooccurcolor = float float float
             | notexistcolor = float float float
             | filename = boolean
             | ratio = boolean | factor = int
             | styleattr
styleattr  ::= color = float float float
             | linestyle = id | marktype = id
             | legend = string
```

Figure 2: HCL grammar

The first kind of information, in lines 1 to 9, gives the values of some global configuration parameters. These describe the layout of relevant files within the file system and some default parameters for the pattern matching tool.

The second kind of information describes the set of projects in which to track pattern occurrences, as illustrated in lines 11 to 20. A project is characterized by the directory (`dir`) containing its source code, by various attributes describing how the project should be represented in graphs, and by the set of studied versions. A version is specified as a tuple, giving the version name, the date when the version was released and the size of the project at that time, expressed in lines of code. Herodotos orders the given versions by date in order to define a successor relation between them.

The third kind of information is the set of code patterns, as illustrated in lines 22 to 28. A pattern is characterized by the name of the file defining the pattern (line 23), some flags to be passed to the pattern matching tool (line 24), and the same set of attributes as in the case of project descriptions to describe how the pattern should be represented in graphs.

The final kind of information is the set of desired graphs. Each graph is described by the name of the file in which to create the graph, a set of attributes and a set of curves. The `xaxis` attribute (lines 31, 46 and 59) defines whether the data is presented by project version (`version`), or by a linear time scale (`date`). The `yaxis` attribute (lines 33, 48 and 61) defines the type of graph to draw. Depending on the type of graph, the user may provide some specific attributes to control its appearance. For instance, `factor` (line 63) may be used with `size` or `density` graphs to change the ordinate scale. Finally, the user gives the set of curves to plot. Each curve is defined by a project, a pattern and a set

```
prefix="tst-config" #output directory
projects="tst-config/projects" #project directory
patterns="tst-config/cocci" #pattern directory
# directory for makefiles
results="tst-config/results"
# pattern finding command
findcmd="spatch %f -sp_file %p -dir %d > %o"
# pattern finding command flags
flags="-timeout 60 -use_glimpse"

project Linux {
 dir = "linux"
 color = 1 0 0
 linestyle = solid
 marktype = circle
 versions = {
  ("linux-2.6.13", 08/28/2005, 4289406)
  [...]
 }
}
[...]
pattern unsigned {
 file = "find_unsigned.cocci"
 flags = "-all_includes"
 color = 0 1 0
 marktype = circle
 linestyle = solid
}

graph gr/hist/vlc-null_ref.jgr {
 xaxis = version
 xlegend = "Versions"
 yaxis = occurrences
 ylegend = "Defects"

 curve project VLC pattern null_ref {
   notexistcolor = 0 0 0
   nooccurcolor = 1 1 1
   color = 0 1 1
   filename = true
   ratio = true
 }
}

graph gr/evol/linux.jgr {
 xaxis = date
 xlegend = "Linux"
 yaxis = sum
 ylegend = "Number of defects"
 legend = "defaults fontsize 8 left"
 project = Linux

 curve pattern unsigned
 curve pattern null_ref
 [...]
}

graph gr/evol/code-size.jgr {
 xaxis = date
 xlegend = ""
 yaxis = size
 ylegend = "Code size\nin MLOC"
 factor = 1000000 # MLOC
 legend = "defaults fontsize 6 x 800 y 3.4"

 curve project Linux
 curve project Wine
 curve project VLC
 curve project OpenSSL
}
```

Figure 3: Herodotos configuration file

of attributes. The project or the pattern may be explicitly given or inherited from the default attributes of the graph.

3.2 Correlating pattern occurrences

Once the pattern occurrences are found in each version of each project using the pattern matching tool, Herodotos correlates identical occurrences across multiple versions. The key challenge in this occurs when changes have been made in the file. The cornerstone of our approach is to compute, for the position of each occurrence in each version, a *prediction* of the position in the successor version. This prediction is computed using the code changes given by `diff`, then compared to what was observed by the pattern matching tool in the successor version. This process automatically correlates most of the pattern occurrences, while the missed correlations are corrected by the user. To ensure the completeness of the correlations, Herodotos proposes a set of possible correlations for the user to annotate.

3.2.1 Automatic correlation

To compute the prediction, Herodotos first uses the GNU `diff` tool to find the differences between a version n and its successor version $n+1$. For a pair of files, `diff` produces a sequence of *hunks*, which are 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. For the i^{th} hunk h_n^i in version n, the position is represented by the tuple $(L_n^{i-}, \delta_n^{i-}, L_n^{i+}, \delta_n^{i+})$ where L_n^{i-} is the starting line number of the hunk i in version n, δ_n^{i-} is the number of lines removed by the hunk in version n, L_n^{i+} is the starting line number of the hunk in the successor version $n+1$, and δ_n^{i+} is the number of lines added by the hunk in the successor version.[1] To simplify the prediction computation, due to a peculiarity of the GNU `diff` tool, we adjust some components of the position tuple as specified by Equations (1) and (2).

$$\text{if } \delta_n^{i-} = 0, L_n^{i-} := L_n^{i-} + 1 \tag{1}$$

$$\text{if } \delta_n^{i+} = 0, L_n^{i+} := L_n^{i+} + 1 \tag{2}$$

We next observe that each pattern occurrence o has a position p_n^o in version n, beginning at line $l_{p_n^o}$. For each position p_n^o, Herodotos computes the index i for which the hunk i is the last one whose starting line is at or before the line containing a given occurrence. In that case, either Equation (3) or (4) holds, denoting an occurrence inside the hunk or after the hunk, respectively. If Equation (3) holds, then the pattern occurs within the lines of code removed by the hunk, and Herodotos considers that the pattern occurrence has been removed. Pattern occurrences that appear in added lines are considered as new. In that case, Herodotos relies on the user to define a possible correlation if there exists at least one uncorrelated occurrence of the pattern in version $n+1$ of the same file.

$$L_n^{i-} \leq l_{p_n^o} < L_n^{i-} + \delta_n^{i-} \tag{3}$$

$$L_n^{i-} + \delta_n^{i-} \leq l_{p_n^o} < L_n^{(i+1)-} \tag{4}$$

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

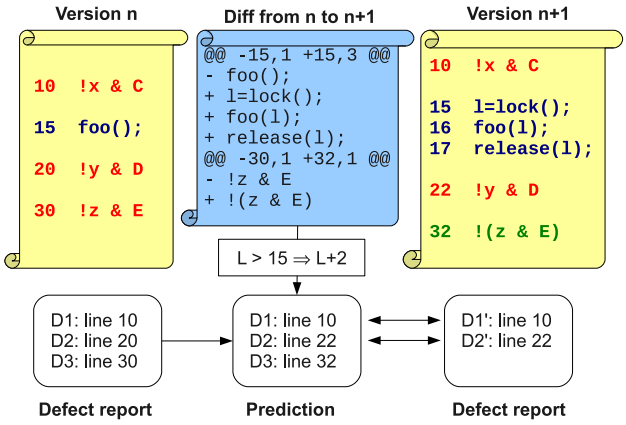


Figure 4: Description of the correlation process

On the other hand, if Equation (4) holds, then the pattern occurrence is after the end of the hunk (if there is no subsequent hunk, $L_n^{(i+1)-}$ is considered to be the line after the end of the file). In this case, the same code is present in both versions of the file. However this code has been shifted by all the modifications performed by the hunks before it. Equation (5) then computes the line $l_{\pi_n^o}$ of the predicted position π_n^o. This equation first computes the offset between the starting line numbers L_n^{i+} and L_n^{i-} of the hunk, which represents a cumulative view of the effects of all the previous hunks, and then computes the difference between the number of lines added and removed by the hunk, δ_n^{i+} and δ_n^{i-}, which represents the local modification of the hunk. Finally, this offset and difference are added to the position of the occurrence in the current version.

$$l_{\pi_n^o} := l_{p_n^o} + (L_n^{i+} - L_n^{i-}) + (\delta_n^{i+} - \delta_n^{i-}) \qquad (5)$$

If a predicted position π_n^o of p_n^o in version $n+1$ is equal to a position $p_{n+1}^{o'}$ reported by the pattern matching tool, then o and o' are considered to be the same occurrence and are correlated across versions n and $n+1$. If no report is found, *i.e.*, the code is unchanged but is no longer considered to be an occurrence, it typically means that some other adjustment to the source code has changed it in such a way that the new code does not match the pattern anymore. For example, when a pattern searches for type errors in the use of a variable, an error may be fixed by adjusting the type of the variable, leaving the reference unchanged.

The correlation process is illustrated by Figure 4 for an arbitrary version n and its successor. In this example, lines 10, 20 and 30 are affected by a common defect, *i.e.*, a nonsensical bit-and operation (`&`).[2] Two changes occur in creating the next version: the defect originally on line 30 is fixed and an unrelated change is performed in line 15. To study such nonsensical operations, we wrote a pattern that detects them. Applying it reports the defects $D1$, $D2$ and $D3$, in version n and defects $D1'$ and $D2'$ in version $n+1$. For each report, Herodotos identifies the relevant hunk using the information provided by `diff`: for $D1$, there is no preceding hunk; for $D2$, the first hunk implies that lines after line 15 now have an offset of $+2$; for $D3$, the second hunk implies that the occurrence may have disappeared. Herodotos then computes a prediction for each report in version $n+1$. In this prediction, $D1$ is still at line 10, and $D2$ is now at lines 22. By comparing the prediction with the real report of version $n+1$, Herodotos infers that $D1$ and $D2$ are still present with names $D1'$ and $D2'$ respectively, and $D3$ has been fixed, as there is no report in that version. If another uncorrelated report were present anywhere in that version, *e.g.*, if constant `E` were changed but parentheses were not added at line 30, Herodotos would have proposed a correlation to the user in the manual phase of the correlation process (phase 4 of Figure 1) described below.

[2]The C operator `!` binds more tightly than `&`, implying that if *e.g.*, `C` is even in Figure 4, then the result is always 0.

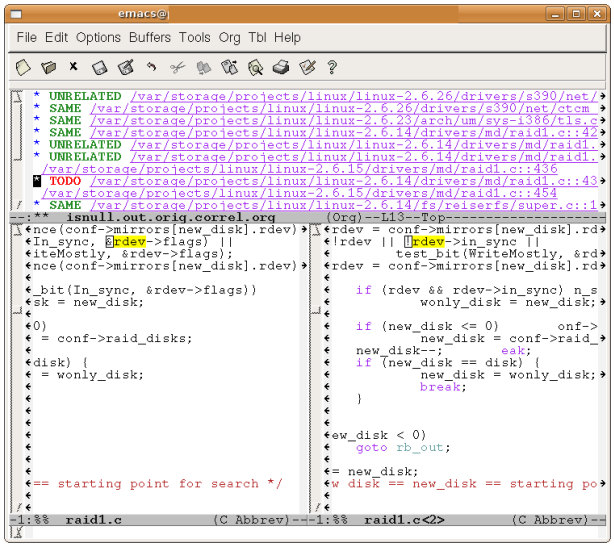


Figure 5: Aiding to provide user correlation hints

3.2.2 *Manual correlation*

When there is a modification within a line containing a pattern occurrence, Herodotos cannot perform the correlation automatically because the occurrence is considered to be part of the code removed in some hunk, even if the occurrence is still present in the file. In this case, Herodotos infers that the occurrence disappears in one version and another occurrence appears in the next one. In this situation, the user must complete the occurrence history. To help the user specify which occurrences should be correlated, Herodotos generates a list of possible correlations consisting of all pairs of possibly related occurrences within a given file. This list, as illustrated in Figure 5, is in the Emacs Org mode format with hyper-links to the source code. In this list, each pair of possibly related occurrences is annotated with a state, either `TODO`, `SAME` or `UNRELATED`, indicating respectively a pair to check, a correlated pair of occurrences, or an unrelated pair. Initially, all pairs have the state `TODO`. For each `TODO` pair, the user can follow the hyperlinks to check the reported occurrences in a version and the next one. If it appears that both occurrences in a `TODO` pair are the same, at possibly different positions, the user changes the state from `TODO` to `SAME`. If the reported occurrences are unrelated, the user changes the state to `UNRELATED`.

As Herodotos proposes a correlation for each pair of occurrences within a given file that are possibly related, the number of proposed correlations is potentially quadratic.

Each occurrence, however, can only be correlated with at most one other. Thus, Herodotos supports an iterative process in which the user annotates some correlations as valid, and then reruns Herodotos to automatically eliminate other correlations that are no longer possible. During this iterative process, Herodotos takes as input not only an occurrence 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`.

4. EVALUATION

We have applied our approach on four open source software projects. In these experiments, we have used the Coccinelle pattern matching tool [6, 31], and a collection of patterns that represent a variety of software defects. These patterns and our results are available at the web site listed at the end of the paper. 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.

4.1 Selected software

To conduct our evaluation, we have selected four software projects with different profiles: Linux [24], Wine [36], VLC [34] and OpenSSL [29]. These cover aspects ranging from a full operating system kernel (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, when available. The releases for the other projects are given in Table 1. There are fewer versions for OpenSSL, because this software was released less often in the considered time period.

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)					
0.9.8	0.9.8a	0.9.8b	0.9.8c	0.9.8d	0.9.8e
0.9.8f	0.9.8g	0.9.8h	0.9.8i	0.9.8j	

Table 1: List of versions used

Figure 6(a) shows the number of lines of C code in each software project across the different versions. This graph was generated by Herodotos, based on the specification in lines 58 to 70 of Figure 3, using the `yaxis` attribute `size`. Linux is the largest software project, with 4 to 6 million lines of C code in the considered time period. Wine is the next largest, with 1 to 1.5 million, and OpenSSL and VLC are the smallest, with 200,000 to 330,000 lines of C code. The `sizepct` graph in Figure 6(b) 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 16%. For VLC, the code size remained essentially the same for a long period, and then spiked, while for the other software projects, the increase has been more linear.

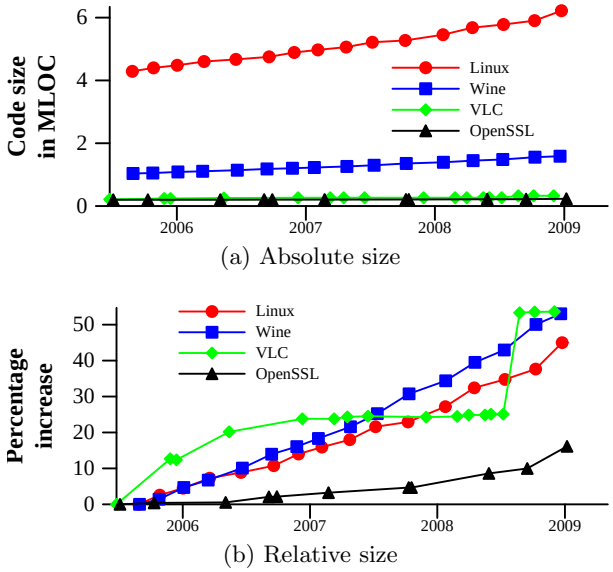


Figure 6: Code sizes of the software versions released between 6/25/2005 and 02/13/2009

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 2: Defect classes

4.2 Selected patterns

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 [7, 23, 27]. A widely cited reference is the Common Weakness Enumeration (CWE) [27] where each weakness is explained and detailed with examples. Table 2 describes a number of entries in the CWE, relating to resource management and code structure.

We have developed patterns to find a range of kinds of defects in the classes shown in Table 2. To allow comparison between the projects, the patterns are project independent. The defect kinds are as follows: 1) A file descriptor is acquired, but not released (`open`), 2) memory is allocated, but not freed (`malloc`),[3] 3) a NULL pointer is dereferenced (`isnull`), 4) a value is dereferenced and subsequently checked for NULL (`null_ref`), 5) a pointer is checked for NULL, when it is already known that it is not NULL (`notnull`), 6) a constant

[3]This rule checks for `kmalloc` and `kfree` for Linux, and `malloc` and `free` for the other projects.

```
1  // notand.cocci
2  @r@ expression E; constant C; position p; @@
3   !@p E & C
4
5  @script:python@ p << r.p; @@
6  cocci.print_main("",p)
```

```
1  // unsigned.cocci
2  @u@ type T; unsigned T i; position p; @@
3   i@p < 0
4
5  @script:python@ p << u.p; @@
6  cocci.print_main("",p)
```

Figure 7: The `notand` and `unsigned` patterns

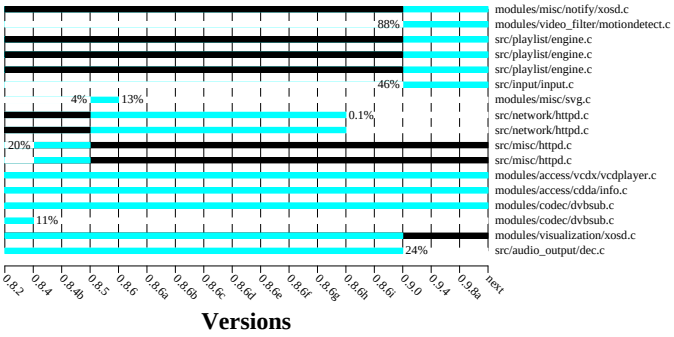


Figure 8: NULL reference defect evolution in VLC

is assigned to a variable, but the variable's value is never used (`unused`), 7) a pointer is compared to 0, rather than NULL (`badzero`), 8) boolean and bit operations are misused (`notand`), and 9) an unsigned value is checked to be less than 0 (`unsigned`).

Examples of these patterns are shown in Figure 7, for `notand` and `unsigned`, which are two of the simpler cases. A pattern consists of a sequence of rules, which may be expressed in either the Coccinelle *semantic patch language*, SmPL, or in Python. Each rule consists of the declaration of a collection of metavariables, *e.g.* `E` and `C`, in the case of the first `notand` rule, followed by either a pattern, expressed using an extension of C code, or code to execute, expressed in Python. Besides metavariables bound to *e.g.*, arbitrary expressions and constants, patterns may also contain *position metavariables*, which are bound to information about the position of the token to which they are attached. In our examples, SmPL code matches a defect occurrence and then Python prints information about the position of each defect occurrence in Emacs org mode format, using appropriate Coccinelle Python library functions.

Some of the considered 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 3 GHz quad-core Xeon processors and 16 GB memory. The combined size of the software projects is 8 GB. 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 processing Linux code. Once the results were generated by Coccinelle, `diff` computes the changes in less three minutes and we used Herodotos to correlate the 22,077 reported defects in about fifteen seconds. We then checked the correlated defects for false positives, and used Herodotos to build the graphical representations and compute the corresponding statistics. This last phase of Herodotos takes less than ten seconds. In the rest of this section, we discuss the usability of Herodotos, present a synthesis of the computed statistics, and finally discuss some threats to the validity of our results.

4.3.1 Usability

Once the pattern occurrences have been generated, there are two points in the execution of Herodotos where the user should intervene: first to provide additional correlations and second to identify false positives. We now consider the amount of work associated with these operations for the patterns and projects in our experiments.

For Linux, Wine, VLC and OpenSSL and the given defect-finding patterns, 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 (pairs of correlated occurrences) between them. 19,371 correlations are performed automatically, while only 282 are proposed to the user for review. Based on this set, and using the iterative process, the user needs to explicitly annotate only 182 correlations; the others are eliminated by Herodotos. Of the manual correlations, 163 of the correlations are annotated as `SAME`, while the remaining 19 are annotated as `UNRELATED`. Herodotos finally generates 2,543 occurrence histories based on the automatic and manual correlations.

Table 3 shows that the correlation process reduces by almost 89% the number of defect reports that have to be checked for false positives. Table 4 compares the number of confirmed defects to the number of reported defects, for each pattern and each project. Except for OpenSSL, the rate of confirmed defects per project is always over 90%. For OpenSSL the rate is 82%, but there are far fewer reports for this project than for the others. The effort required to check the false positives depends on the properties of the kind of defect being detected, and is independent of Herodotos. For example, `notand`, which was illustrated in Figure 4, requires only checking the local code structure, while `malloc` requires checking multiple lines of code to ensure that the control-flow path that is detected as missing a `free` is feasible during execution.

4.3.2 Graphs and statistics

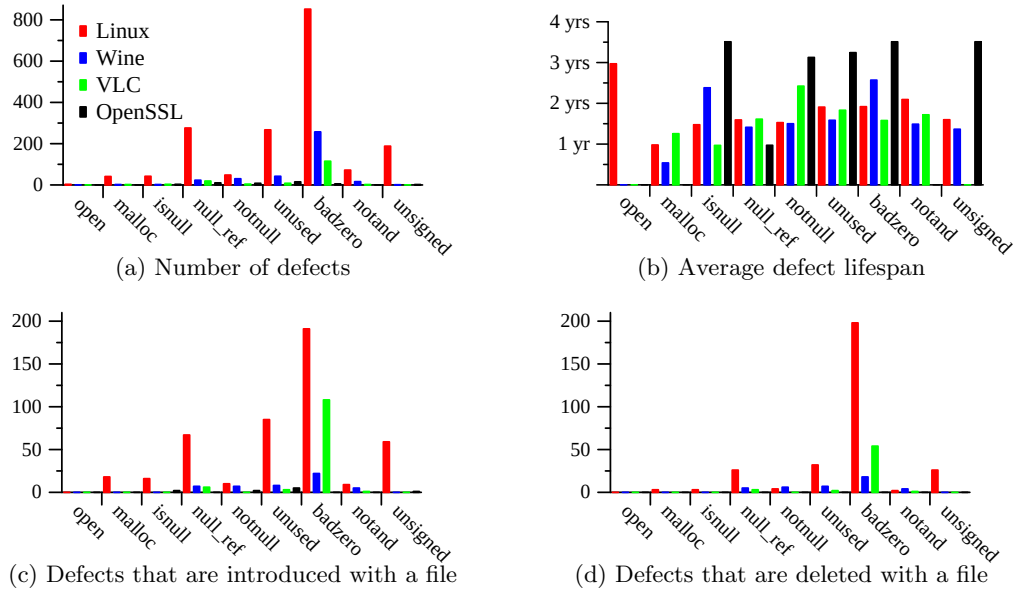
Figures 8, 9, and 10 illustrate the graphs produced by Herodotos. When generating these graphs, Herodotos also prints the calculated values on the standard output, allowing other forms of processing of the same information. Figure 8 shows an `occurrences` graph, illustrating the evolution of the matches of a specific pattern, `null_ref`, in a specific project, VLC. Figure 9 shows various bar graphs, representing the number of occurrences of each kind of defect at any point in the considered time period for each software

	Reports	Automatic correlations	Missing correlations	Correlations proposed to the user	User provided correlation annotations		Occurrence histories
Linux	16,103	14,050	121	176	130	0.92%	1,932
Wine	4,281	3,864	22	79	27	0.69%	400
VLC	1,337	1,175	3	3	3	0.25%	159
OpenSSL	356	282	17	24	22	7.24%	52
Total	22,077	19,371	163	282	182	0.93%	2,543

Table 3: Correlation effectiveness

Defect kinds		Software projects: Linux C. / R.	Wine C. / R.	VLC C. / R.	OpenSSL C. / R.
Resource	File descriptor not released (`open`)	3 / 4	0 / 0	0 / 0	0 / 1
	Memory not released (`malloc`)	42 / 44	2 / 2	2 / 2	0 / 0
	Dereference after NULL (`isnull`)	42 / 92	2 / 5	3 / 5	3 / 4
	Dereference before checking NULL (`null_ref`)	276 / 309	23 / 33	19 / 22	10 / 13
Useless code	Double check a pointer with NULL (`notnull`)	48 / 69	30 / 30	4 / 4	8 / 10
	Assign a constant to an unused variable (`unused`)	267 / 286	42 / 49	8 / 9	15 / 17
Insecure code	Compare with zero instead of NULL (`badzero`)	852 / 863	257 / 264	115 / 115	5 / 5
Erroneous code	Wrong use of ! with & (`notand`)	72 / 76	16 / 16	2 / 2	0 / 0
	Check if an unsigned value is less than 0 (`unsigned`)	188 / 189	1 / 1	0 / 0	2 / 2
	Total	1,790 / 1,932	373 / 400	153 / 159	43 / 52

Table 4: Confirmed (C.) and reported (R.) defects by defect category



(a) Number of defects

(b) Average defect lifespan

(c) Defects that are introduced with a file

(d) Defects that are deleted with a file

Figure 9: Generated statistics for each defect kind and each software project

project (`cumulsum`), the average lifetime of each such defect (`avglifespan`), the number of such defects that have been introduced when adding a new file, and the number that have been removed when deleting a file. Finally, the `sum` graphs in Figure 10 show the evolution in the number of defects throughout the studied period.

We now present these results from several points of view: in terms of a particular pattern and project, in terms of the projects, in terms of the patterns, and in terms of the evolution of the defects through the project history.

Per pattern and project.

Figure 8 represents the `occurrences` graph, defined by lines 30 to 43 of Figure 3, for the VLC project and the `null_ref` defect pattern. This kind of graph shows the lifetime of each defect, the lifetime of the file in which it occurs and the percentage of the lines of code that change when the defect is introduced and removed. Determining the lifetime of each defect requires the correlation of defect reports over multiple versions, as performed by Herodotos. For a given pattern and project, there is a light blue/grey bar (Figure 3, line 39) running from the version in which the pattern was introduced to the version in which it disappeared. A black bar (line 37) indicates the range 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 8, the first defect was introduced when its containing

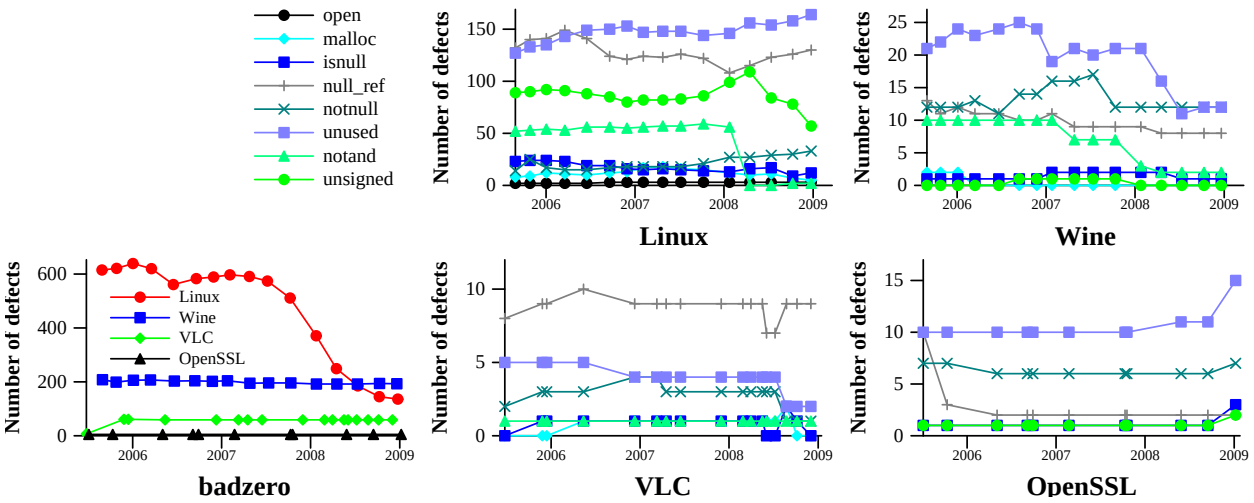


Figure 10: Generated graphs representing each defect evolution for each software project

file was added to the software project, while the second defect was the result of a modification in the existing code. In this case, the file was substantially rewritten, by more than 80%. 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.

When studying defective code patterns, the `occurrences` information can help in determining the possible conditions that lead to defects. Indeed, popular wisdom has it that introducing new code and new features also introduces new defects. This theory is substantiated for VLC for the given defect type, where the versions that introduce defects also typically contain many changes.

Based on the same information, Herodotos can also generate `birthfile` and `deathfile` graphs (not shown), showing how many occurrences are introduced or removed with the file to which they belong, at each version. These kinds of graphs illustrate the code quality of new and obsolete services, when they are respectively added and removed.

Per software project.

As shown in Table 4, Linux, which is 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. Wine has many similarities with Linux. As shown in Figure 6(b), both projects have been growing at a similar rate. Wine also has 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 code size. Finally, OpenSSL confirms its position as stable security software. Indeed, as shown in Table 4, it has the lowest number of confirmed defects. The developers of OpenSSL are also very conservative, which leads to long defect lifetimes, as shown in Figure 9(b).

Per defect category.

Figure 9 presents the statistics computed by Herodotos: the number of defects, the average lifespan, and the number of defects introduced and removed at the same time as the file they belong to. Of these statistics, calculating the number of defects and the average lifespan relies on the correlation of defect reports over multiple versions, as performed by Herodotos. The defect kinds are presented in the same order as in Table 4.

Despite the fact that many static and dynamic analysis tools have considered defects related to the release of allocated memory, Figure 9(a) shows that many such errors still exist, particularly in Linux code, where user-space tools such as Valgrind [28] cannot be used. These defects, however, tend to have a shorter lifespan, as shown in Figure 9(b), particularly for `malloc` in Wine and `isnull` in VLC.

OpenSSL defects tend to have a long lifespan in general. Depending on the project, other defects may have a long lifespan such as deferencing of a NULL value (`isnull`, in Wine), redundant NULL tests (`notnull`, in VLC), and comparing a pointer to zero (`badzero`, in Wine). Of these, only `isnull` directly leads to a program crash, although `notnull` may represent a test of the wrong value, which can lead to a crash in the access to some other data. Except for Linux, in which such defects have a relatively short lifespan, the number of such defects is small. The remaining defects tend to have a lifetime between a year and a half, and two years.

The defect of comparing a pointer to zero is common in newly added files (Figure 9(c)) and has a long lifespan (Figure 9(b)), with the defect often being either still present or removed only when the file disappears (Figure 9(d)).

Evolution through the project history.

Figure 10 shows the `sum` graphs generated by Herodotos. For each project and each kind of defect, a line gives the evolution of the number of defects. The lines related to `badzero` are plotted in a separate graph to improve readability.

For Linux, we observe that the reported defects about misuse of boolean and bit operators (`notand`) and comparison of unsigned with zero (`unsigned`) have declined dramatically since early 2008. As we have been using Coccinelle for finding and fixing bugs in the Linux kernel since around this time, it could explain in part the trend we observe on the later studied versions. However, we still find some defects and it

would be interesting to learn why they have not been fixed. Has a patch already been submitted? If so, why it is not incorporated? 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 many unused variables, many of which have been cleaned up in 2008. We also note that misuse of boolean and bit operators (`notand`) has declined significantly. This improvement began 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, with respect to its size. However, unused variables have recently been 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, suggesting that developers should focus on these defects, and should keep these issues in mind when adding new code. Other kinds of defects are stable over time.

Defects in OpenSSL are stable, suggesting that little effort has been made to fix them. The number of unused variable defects has even risen recently. However, OpenSSL is also the project with the lowest rate of growth, around 16% (Figure 6(b)), which suggests that its development is not very active even though it is widely used.

4.4 Threats to validity

To assess the threats to the validity of the study, we consider issues related to the correlations performed by Herodotos, and then issues related to the choice of projects and the pattern matching process.

Correlations.

We currently have a fairly restrictive notion of *identical* occurrences across versions. Our approach does not deal with variable, file and directory renaming, which may lead to an artificially high occurrence turnover, *i.e.*, occurrences removed in one version and added to the next one. To address this issue, we can define a more general definition of identical occurrences. Nevertheless, the best definition might be specific to the pattern studied and the scale, *i.e.*, variable, file or directory renaming. Defining what is an identical occurrence across versions thus remains an open question.

We have not systematically checked the automatic correlations for false positives, as the large number of reports makes this infeasible. Nevertheless, the automatic correlations by definition refer to blocks of code that have not changed between the two versions, and thus we believe that false positives are unlikely.

Representativeness of the software projects.

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

The nine patterns used are project independent. Generic defects may, however, not be representative of all of a project's defects. Mechanisms to infer software-specific defects [21, 22] can allow studying software-specific defect histories.

To prevent biased results, the user must carefully choose the program matching tool, the projects and the occurrence patterns. The manual verification of the defects found in the occurrence histories may suggest to the user that the choice of tool or pattern is inappropriate, if there are many false positives.

Static pattern matching.

The use of pattern matching against the source code rather than information collected at run time means we may have some false positives in the defect reports when the pattern involves disjoint code fragments that may turn out not be connected by any actual execution path. We have tried to remove these false positives, but some may remain. Moreover, the defects we are looking for must match the provided patterns. In some cases, such as `malloc`, the pattern is fairly restrictive, to prevent false positives. This strategy may nevertheless lead to false negatives. Finally, we have primarily relied on our own expertise to distinguish true defects from false positives, although for each defect type, we have submitted and had accepted a number of patches to the Linux kernel or noted patches related to the defect type that have been submitted by others.

5. RELATED WORK

Previous work has considered either static code analysis to find defects [3, 7, 8, 9, 35] or defect history based on bug trackers [13, 25], but little has been done to build tools to track defects over time in the presence of code modifications. Chou *et al.* [7] 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.* [23] 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.

Other work [9, 22, 35] based on static code analysis has been used for finding defects in upcoming or recent releases, but without consideration of a long period of time to build a defect history. Defect history has, however, been studied by coupling a source code management (SCM) system, generally CVS, with a bug tracking system. DynaMine [25] applies data mining techniques to the data collected by an SCM system to find frequent application-specific coding patterns that can be used to check for bugs. ROSE [37] uses the same technique to suggest modification sites during software evolution. Finally, iBUGS [10] 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 pattern matching against the source code.

Recent work has also considered the automatic correlation of defects, or warnings. Spacco *et al.* [33] use the warning message from a bug finding tool as an identifier with which to compute a hashcode. Identical defects are assumed to have the same hashcode, allowing them to construct a defect history. The problem is that according to the elements taken into account, "identical" defects may greatly vary. In contrast, our approach is based on the exact position in the file, and its evolution. Kim and Ernst [20] and Śliwerski *et al.* [32] rely on the log messages developers provide in an SCM system to identify defects. These approaches require

access to the SCM system and assume that developers use a consistent set of keywords to characterize each commit. In the former approach, every line modified by a commit has the same status as the others, whether or not it is bug-related. Extraneous modifications, such as removal of trailing spaces, may thus cause a line to be incorrectly annotated as a bug fix. This incorrect annotation is then back ported to the previous versions. Finally, the authors do not explain how the line status is propagated beyond the first previous revision, when other changes have to be taken into account. The latter approach suffers from the same problem when a syntactic modification does not change the semantics of the program. Boogerd and Moonen [5] use a history collecting mechanism based on that of Spacco and of Kim and Ernst. Their approach thus suffers from the same strong dependency on the SCM system.

Tracking occurrences has also been applied to the study of code clones. Duala-Ekoko and Robillard [11] propose a similar approach to ours for tracking clones. It is based on a clone detector and an abstract representation of clone occurrences. But their tool is tightly coupled with a particular clone detector. Moreover, they relax the definition of clone position with heuristics to improve clone detection, which weakens the tracking possibility. Kim *et al.* [19] study clone genealogies using a clone detector and a location tracker. Their tool identifies multiple correlations when there is ambiguity between multiple clones, when one has a more similar structure but the other is located at a more expected place. We instead allow the user to provide additional information, and provide assistance for this task.

In our experiments, we have used the standard GNU `diff` tool [26], as it is widely available. However, the Patience diff algorithm [1, 4] or some other element matching tool [16] may lead to a better correlation. We will investigate this in future work.

Interesting patterns to study can be discovered with tools that infer structural changes, such as spdiff [2] for C code, or LSdiff [17, 18] for Java code. These tools infer abstract descriptions of changes, while Herodotos tracks persistent pattern occurrences. However, they may be used in conjunction with a pattern finder to generate input for Herodotos.

6. CONCLUSION

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

In future work, we are considering how to exploit more information from SCM systems and word-based diff to improve the automatic correlation process. For instance, Herodotos is currently not able to correlate pattern occurrences when a file is renamed or moved to another directory. The git SCM system [15] tracks the repository content as a whole, and it is thus able to infer file and directory renaming. `wdiff` [14] works at the granularity of words, rather than lines, and could be used for in-hunk occurrences. Using the information provided by these tools could help Herodotos infer more correlations. In the specific case of defects, 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 defects have been found and fixed. Finally, the Coccinelle pattern matching tool allows the user to define software-specific patterns [22]. We plan to exploit this feature to study software-specific defects and look for new defect categories from a software-specific point of view.

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

Availability: The Herodotos tool and data from this paper are available at `http://www.diku.dk/~npalix/herodotos/`

Acknowledgements: We would like to thank the AOSD reviewers and especially Erik Ernst for helpful feedback on this paper. This work has been supported in part by the Danish Research Council (grant 274-08-0214), and the French ANR blanc project NT09_487593.

7. REFERENCES

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

[2] J. Andersen and J. Lawall. Generic patch inference. In *23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 337–346, L'Aquila, Italy, Sept. 2008.

[3] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *The first ACM SIGOPS EuroSys conference (EuroSys 2006)*, pages 73–85, Leuven, Belgium, Apr. 2006.

[4] S. Bespamyatnikh and M. Segal. Enumerating longest increasing subsequences and patience sorting. *Information Processing Letters*, 76(1-2):7–13, Nov. 2000.

[5] C. Boogerd and L. 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, Sept. 2008.

[6] J. Brunel, D. Doligez, R. R. Hansen, J. Lawall, and G. 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, Jan. 2009.

[7] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, pages 73–88, Banff, Canada, Oct. 2001.

[8] P. Cousot, R. Cousot, J. Feret, A. Miné, L. Mauborgne,

D. Monniaux, and X. 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.

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

[10] V. Dallmeier and T. 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, Nov. 2007.

[11] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 158–167, Minneapolis, USA, May 2007.

[12] D. R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Fourth USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, San Diego, CA, Oct. 2000.

[13] M. Fischer, M. Pinzger, and H. 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, Sept. 2003.

[14] Free Software Foundation Inc. wdiff: comparing files on a word per word basis. `http://www.gnu.org/software/wdiff/`.

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

[16] M. Kim and D. Notkin. Program element matching for multi-version program analyses. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 58–64, Shanghai, China, May 2006.

[17] M. Kim and D. Notkin. Discovering and representing systematic code changes. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 309–319, Vancouver, Canada, 2009. IEEE Computer Society.

[18] M. Kim, D. Notkin, and D. Grossman. Automatic inference of structural changes for matching across program versions. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 333–343, Minneapolis, MN, USA, 2007. IEEE Computer Society.

[19] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 187–196, Lisbon, Portugal, Sept. 2005.

[20] S. Kim and M. 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, Sept. 2007.

[21] J. Lawall, G. Muller, and N. 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, Mar. 2009.

[22] J. L. Lawall, J. Brunel, R. R. Hansen, H. Stuart, G. Muller, and N. 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.

[23] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. 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, USA, 2006.

[24] Linux kernel. `http://kernel.org/`.

[25] B. Livshits and T. 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, Sept. 2005.

[26] D. MacKenzie, P. Eggert, and R. Stallman. *Comparing and Merging Files With Gnu Diff and Patch*. Network Theory Ltd, Jan. 2003.

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

[28] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 89–100, San Diego, CA, USA, 2007.

[29] OpenSSL. `http://www.openssl.org/`.

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

[31] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in Linux device drivers. In *EuroSys 2008*, pages 247–260, Glasgow, Scotland, Mar. 2008.

[32] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, Saint Louis, MO, USA, May 2005.

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

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

[35] D. Wheeler. Flawfinder home page. Web page: `http://www.dwheeler.com/flawfinder/`, Oct. 2006.

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

[37] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.