1. **An Exploratory Study of the Evolution of Software Licensing:**

Free and open source software systems (FOSS) are distributed and made available to users under different software licenses, mentioned in FOSS code by means of licensing statements. Various factors, such as changes in the legal landscape, commercial code licensed as FOSS, or code reused from other FOSS systems, lead to evolution of licensing, which may affect the way a system or part thereof can be subsequently used. Therefore, it is crucial to monitor licensing evolution. In this paper, the authors proposed an approach to automatically track the changes occurring in the licensing terms of a system.

If a company develops a product that incorporates some FOSS components/applications, the developers would have to carefully analyze the overall licensing compatibility of all the included components to update even a single component. This compatibility analysis is usually done manually, or semi-automatically, by verifying that all bundled source files and binaries have been released under compatible licenses. Unfortunately, this analysis is not a one-time activity; each modification to any bundled component may involve licensing statements and thus impact its use/integration.

The license (or licenses) under which a file is made available is usually contained inside blocks of comments at its beginning which were referred to as the *licensing statement(s)* of a file. A change to the licensing statement might be a change to the name of the license (when the statement refers to its name) or a change to the license itself (when the license is in the statement). This paper proposed an approach to automatically track the licensing evolution of systems, identifying changes in licenses and copyright years. The study revealed that licensing evolution is a frequent and relevant phenomenon in many systems and that, while FOSS developers are concerned with licensing issues, they manage, evolve, and update licensing statements in different ways.

This paper proposed a method to track the evolution of software licensing and investigated the relevance of licensing evolution on six FOSS systems. Most noticeably, the authors observed license changes, from one license to another, license additions, *e.g.*, files without license were updated with a license, and license modifications. For large systems like Eclipse-JDT, Mozilla, or the BSD kernels, the amount and frequency of licensing changes would make difficult their manual analysis, thus highlighting the usefulness of an automatic analysis method. Finally, the authors investigated changes occurring to copyright years and found that they are updated to protect new code when substantial changes are made to a source code file.

1. **Identifying Licensing of Jar Archives using a Code-Search Approach**

Free and open source software strongly promotes the reuse of source code. Some open source Java components/ libraries are distributed as jar archives only containing the bytecode and some additional information. For whoever wanting to integrate this jar in her own project, it is important to determine the license(s) of the code from which the jar archive was produced, as this affects the way that such component can be used. This paper proposed an automatic approach to determine the license of jar archives, combining the use of a code-search engine with the automatic classification of licenses contained in textual files enclosed in the jar.

A Java application runs virtually on any known software and hardware architecture without the need of recompilation. Java promotes the reuse adoption via componentization and classes bundled into jars (pre-compiled versions of the components). However, one of the major challenges of reusing a component (in source code or binary form–such as a jar) is to know its provenance: who its copyright owner is, and more important, under which license it is made available. In essence, when a jar file is downloaded, it might not be clear where its corresponding source code can be found, and under which license the component is made available. Thus it might not be easy to determine if such jar can or cannot be legally used. This paper proposed an approach to automatize the license identification process supporting developers in the cumbersome mining activities required to discover the license of the source code included in a jar.

The automatic license mining approach of this paper infers the license of jar archives and their contained classes. This approach combines various sources of information. First, it uses information decompiled from the bytecode of its classes to query a code search engine, with the objective of finding the classes and packages contained in the jar, and retrieving the license as classified by code search engines (we call this the inferred license). Also, it mines the textual files contained in the jar, automatically classifying licenses contained into these files using an existing classifier, in order to verify whether the license of these textual files (which we call the declared license) is consistent with the licenses inferred for the class bytecode. This approach was validated over 37 jars belonging to 17 different projects. Results indicated that the proposed approach ensured a high percentage—on average 95%–of correct license classification, although it had limitations in the capability of retrieving licenses for specific versions of a class, which can be a cause of imprecision when licenses change over the time, and that, very often, the textual license provided with the jar is not fully consistent with those of the source code of the of classes contained in the archive.

1. **Automatic Extraction of a WordNet-like Identifier Network from Software**

Softwares are designed to be used a significant amount of time, therefore maintenance represents an important part of their life cycle. It has been estimated that a lot of the time allocated to software maintenance is spent on the program comprehension. Many approaches using the program structure or external documentation have been created to ease the program comprehension. However, another important source of information is still not widely used for this purpose: the identifiers. In this article, the authors proposed an approach, based on Natural Language Processing techniques, that automatically extracts and organizes concepts from software identifiers in a WordNet-like structure: lexical views. Those lexical views give useful insight on an overall software architecture and can be used to improve results of many software engineering tasks.

Softwares are designed to be used a significant amount of time, therefore maintenance represents an important part of their life cycle. It is estimated that up to 60% of the time allocated for software maintenance is spent on the comprehension of the program. A lot of work has been done to recover important insight about a program by the analysis of its structure or its external documentation. Unfortunately, a fundamental other source of information present in softwares is still not widely used: the identifiers. The identifiers are short names given by the developers to the different elements (functions, classes, attributes, types, variables) defined in a software. They constitute about 33% of all tokens in the source code of softwares. They also have a strong impact on the program comprehension.

Disposing of a lexical network like WordNet is very useful to navigate through a set of words. Moreover, it can be used as the basis to compute several semantic similarity measures. In such a lexical network, words are grouped into sets of words having the same meaning (synsets) and connected by several distinct relations (like is more general than or is a part of). The main problem is that obtaining such a lexical network usually requires a lot of manual work done by many domain experts.

In this paper, the authors proposed a novel approach that automatically classified a set of identifiers in a WordNet-like structure. They called this structure a lexical view. It highlighted the hierarchical relations between the identifiers. Moreover, it included several implicit concepts that have been automatically extracted from the initial set of identifiers. This approach, that adapts and uses techniques from the Natural Language Processing (NLP) field, performed the following successive steps:

* cut up identifiers in order to find the primitive words they are composed of,
* classify the previously extracted primitive words into lexical categories (noun, verb, adjective, . . . ),
* apply rules specific to the English language to determine which words are dominant and impose the meaning of the identifiers,
* extract implicit important words,
* Organize the initial identifiers together with the freshly extracted words in a WordNet-like lexical view.

The contribution presented in this paper is a fully automated approach that extracts from those identifier names the main concepts of a program, and that organizes them into a lexical view making explicit the relations linking them. The main benefits of this approach are:

* It is fully automated,
* It is language- and paradigm-independent,
* It can be applied to any piece of code, of any granularity, from a single class to an entire system, in order to analyze any kind of identifiers (class names, function names, attributes names),
* It extracts the concepts explicitly included in the identifiers, as well as those implicitly included,
* It organizes the concepts in a lexical view allowing to navigate the concepts through their hierarchical structure.

1. **Identifying Security Bug Reports via Text Mining: An Industrial Case Study**

A bug-tracking system contains bug reports (BRs) collected from various sources such as development teams, testing teams, and end users. When bug reporters submit bug reports to a bug-tracking system, the bug reporters need to label the bug reports as security bug reports (SBRs) or not, to indicate whether the involved bugs are security problems. These SBRs generally deserve higher priority in bug fixing than not-security bug reports (NSBRs). However, in the bug-reporting process, bug reporters often mislabel SBRs as NSBRs partly due to lack of security domain knowledge. To address this important issue, the authors developed a new approach that applies text mining on natural-language descriptions of BRs to train a statistical model on already manually-labeled BRs to identify SBRs that are manually-mislabeled as NSBRs.

If the model classifies an SBR as an NSBR, or if the model classifies an NSBR as an SBR, then the result is a misclassification. The success rate of the model is the number of correct classifications divided by the total number of classifications.

This approach identified a high percentage (78%) of SBRs mislabeled as NSBRs by bug reporters for a large Cisco software system. To increase the accuracy of our model, the authors suggested that the software engineers should retrain the model when there are new SBRs being verified by security engineers. But the trained model was not recommended to be applied to software systems in which the SBRs describe different types of security bugs than those that were used to train the model. In summary, our approach effectively automates the identification of SBRs that would otherwise require substantial efforts by security engineers to manually assess each BR in a BTS to determine which BRs are SBRs.

1. **A sentence-matching method for automatic license identification of source code files**

The reuse of free and open source software (FOSS) components is becoming more prevalent. One of the major challenges in finding the right component is finding one that has a license that is adequate for its intended use. The license of a FOSS component is determined by the licenses of its source code files. In this paper, we describe the challenges of identifying the license under which source code is made available, and propose a sentence-based matching algorithm to automatically do it. We demonstrate the feasibility of our approach by implementing a tool named *Ninka*.

One of the major challenges of intellectual property clearance is to identify the license under which a FOSS component, and each of its files, is made available. This is due to several factors:

* There is a vast number of open source licenses, some approved by the Open Source Initiative and many more that are not
* A FOSS product might be made available under several licenses,
* Different versions of a FOSS component might be available under different licenses; and
* The overall license of a product might be different than the license of each of its files.

The contributions of this paper are:

* The authors described and categorized the challenges of license identification.
* They proposed a method for license identification based on the analysis of each of the sentences in the license statement of a source code file
* They performed an empirical evaluation of our method and three other license identification tools showing that this method outperforms the others in precision and speed.

The authors empirically evaluated *Ninka* and other similar tools and found that *Ninka* is better and faster at identifying the license name and version than these tools, at the cost of lower recall. They also performed an empirical study of a large collection of free and open source applications and discovered, for example, that the GPLv2+ is the most commonly used license in terms of number of files and number of applications that use it. It was also discovered that licensing statements are prone to errors. The integration of license verification tools into development environments will alleviate this problem.