Redacting Sensitive Information in Software Artifacts

Mark Grechanik
U. of Illinois at Chicago
Chicago, IL 60607, USA
drmark@uic.edu

Collin McMillan
University of Notre Dame
Notre Dame, IN 46556, USA
cmc@nd.edu

Tathagata Dasgupta
U. of Illinois at Chicago
Chicago, IL 60607, USA
tdasgu2@uic.edu

Denys Poshyvanyk
College of William and Mary
Williamsburg, VA 23185, USA
denys@cs.wm.edu

Malcom Gethers
University of Maryland, Baltimore County
Baltimore, MD, 21250, USA
mgethers@umbc.edu

ABSTRACT

In the past decade, there have been many well-publicized cases of source code leaking from different well-known companies. These leaks pose a serious problem when the source code contains sensitive information encoded in its identifier names and comments. Unfortunately, redacting the sensitive information requires obfuscating the identifiers, which will quickly interfere with program comprehension. Program comprehension is key for programmers in understanding the source code, so sensitive information is often left un-redacted.

To address this problem, we offer a novel approach for *REdacting Sensitive Information in Software arTifacts (RESIST)*. RESIST finds and replaces sensitive words in software artifacts in such a way to reduce the impact on program comprehension. We evaluated RESIST experimentally using 57 professional programmers from over a dozen different organizations. Our evaluation shows that RESIST effectively redacts software artifacts, thereby making it difficult for participants to infer sensitive information, while maintaining a desired level of comprehension.

Categories and Subject Descriptors

D.2.5 [**Software Engineering, Testing and Debugging**]: Testing tools; D.2.9 [**Software Engineering, Management**]: Productivity

General Terms

Algorithms, Experimentation, Documentation, Management

Keywords

privacy, redaction, associative rules, program comprehension

1. INTRODUCTION

Creating and maintaining software is beset by many challenges, which include protecting sensitive information. Not only do recent data protection laws and regulations around the world prohibit organizations from disclosing confidential data [51, 13, 56], but also it is embarrassing for these organizations to accidentally release sensitive information in software artifacts. In the past decade, there have been many well-publicized cases of leaking source code that contains sensitive information from different well-known companies including Cisco, Facebook, and Microsoft, Symantec [16, 34, 29, 50, 36, 35, 48, 62, 26]. In the latter case that is also the most recent one, an anonymous hacker demanded that Symantec pay $50,000 to prevent the release of the source code for PCAnywhere, and when Symantec refused, the source code was released to open-source repositories revealing sensitive information related to trade and technical secrets. In case of Microsoft, sensitive information about Microsoft partners was revealed in the leaked source code. Clearly, sensitive information should be redacted in source code and other software artifacts, however, doing it manually in more than 28MLOC of Microsoft source code that was leaked is a very difficult exercise. More importantly, blindly removing sensitive information from software artifacts may severely reduce program comprehensibility thereby aggravating different software maintenance and evolution tasks.

1.1 Sensitive Information in Software Artifacts

To protect and hide sensitive information, many companies have policies and rules about redacting sensitive information in business documents – it is well-known problem, since one in three companies investigates breaches of sensitive data at least once a year [60]. For example, NASA published detailed instructions to its employees on removing sensitive information from Microsoft Word and PDF documents [42]. Essentially, business analysts and security experts redact business documents by identifying words that enable attackers to infer sensitive information and by replacing these words with blanks or random characters. In this paper, we will call these words *sensitive words*. Even though there are solutions that partially automate the redaction process [18, 15, 52], sensitive information is still leaked as the recent case with Transportation Security Administration shows, where sensitive airport screening procedures were leaked as a result of the failure to properly redact documents [61]. As difficult as it is to redact plain text documents, there are no solutions for redacting sensitive information in software artifacts.

Redacting sensitive information in business documents is often detrimental to software development. Removing sensitive words from business and requirements documents leads to ambiguous and misunderstood requirements, which often leads to project failures [33, pages 365-394]. Software engineers need the all information that they can get to create software applications, that is, to translate information from these documents (i.e., the problem domain) into the solution domain (i.e., the domain in which engineers use their ingenuity to solve problems [31, pages 87,109]). If managers face a chance that a software project may fail because engineers build incorrect application due to redacting important information in documents, these managers eliminate this risk by giving software engineers full access to all business documents. A standard procedure is for all stakeholders to sign a non-disclosure agreement, which is a legal document designed to prevent disclosure of sensitive information under the threat of legal actions against violators.

Unfortunately, software engineers often encode sensitive words from business and requirements documents into source code and different design and testing artifacts. Four of the most common reasons are that these engineers knowingly break compliance rules to get their job done, meet a critical deadline, are unaware they were breaking a compliance rule, or they do not care about compliance rules [60]. One way or the other, software engineers encode sensitive words into comments, the names of classes, fields, methods, variables, and packages, program constants, configuration files, or environment variables.

Since source code and other related software artifacts are considered intellectual property, it was not a problem for companies to have sensitive information in these artifacts. However, this situation has radically changed in the past decade. Source code can now be easily leaked by uploading it to an open-source software repository, from where it can be downloaded and examined by hundreds of thousands of software engineers from all over the world. Unauthorized disclosure of source code is embarrassing, and the damage is multiplied when the code reveals sensitive information [35].

Moreover, organizations and companies that develop and own applications often hire external service providers to provide various services including but not limited to supporting customers who use these applications, writing technical and user manuals, performing corrective and perfective maintenance of the source code, and testing applications. Globally distributed software development is common for the majority of software projects, where third party service providers are hired to facilitate various project-related activities.

It is only in the past decade that applications are increasingly being tested by third-party specialized software service providers, which are also called test centers. Numerous test centers have emerged and often offer lower cost and higher quality when compared to in-house testing. The test outsourcing market is worth more than USD 25 billion and growing at 20% annually, making it the fastest growing segment of the application services market [4, 20]. In general, test centers are not authorized to access sensitive information of their client companies. However, to perform white and gray-box testing, these test centers must obtain access to the source code, and having sensitive information in this code can inadvertently result in violating different privacy laws as well as revealing trade secrets. Outsourcing white-box and gray box testing is a recent phenomenon, and many testing outsourcing companies (e.g., STC Third Eye[1] and DeeCoup Solution[2]) provide these services.

1.2 A Fundamental Problem

A fundamental problem at the intersection of software development, distributed service providing, and data privacy is how to allow an application owner to release its source code and other software artifacts to different service providers with guarantees that sensitive information is removed from the source code and these artifacts while preserving program comprehension, which is important for the job functions of these service providers. Consider test center engineers whose job is to construct acceptance tests to determine if programmers implemented requirements correctly. When creating these tests using white and gray box testing, it is important for test center engineers to understand the source code so that they can determine how requirements are implemented. Otherwise, if these engineers cannot understand the source code, they cannot develop effective tests. Similar argument applies to other activities of external service providers.

A brute-force solution to redacting sensitive information is to replace all sensitive words in the source code with randomly generated strings. This approach has a major drawback – it replaces meaningful and descriptive names that programmers choose with random sets of characters thereby destroying program comprehensibility. Descriptive names are crucial to understand code [49]; in general, understanding code and determining how to use it, is a manual and laborious process that takes anywhere from 50% to 80% of programmers' time [17, 21]. This brute-force solution ensures full redaction of sensitive information, however, it destroys program comprehension to a large degree. Finding a solution that balances the goals of privacy and program comprehension by obfuscating sensitive information is a goal of this paper.

Naturally, different applications have different privacy goals and levels of data sensitivity – privacy goals are more relaxed for applications that manage movie ticket databases than for applications that are used within banks or government security agencies. Applying more relaxed redaction to applications is likely to result in greater program comprehension since fewer sensitive words are replaced; conversely, stricter protection makes it more difficult for service providers to do their job that requires good program comprehension. The latter is the result of two conflicting goals: leaving programmers' intent as transparent as possible and hiding sensitive information from external parties (i.e., attackers) who need this transparency to accomplish their jobs. Balancing these goals, i.e., to redact sensitive information while preserving program comprehensibility is emerging to be an important problem.

1.3 Our Contributions

To address this issue, we offer a novel approach for automatically *REdacting Sensitive Information in Software arTifacts (RESIST)* that combines an entropy-based data privacy metric, associative rule mining for finding replacement words that can conceal sensitive information, program comprehension metrics that are based on the notions of conceptual cohesion and coupling [41, 46], and a sound renaming algorithm for Java programs [49]. With RESIST, organizations can balance the goals of preserving program comprehensibility while releasing applications to external service providers with a controlled disclosure of sensitive information. A movie showing how to use RESIST is also available[3]. Our work is

[1]http://www.stcthirdeye.com/white-box-testing.htm

[2]http://www.deecoup.com/solutions/testing-outsourcing

[3]http://www.youtube.com/watch?v=FuARbg_4ACM

```
Set orderItemSeqIdCompleted = FastSet.newInstance();//for items ... after invoicing
Set workEffortIdCompleted = FastSet.newInstance();//for work efforts ... after invoicing
                                                   //(this service supports outsourced tasks)
```

Figure 1: A motivating example that shows how sensitive information about outsourcing manufacturing is encoded in the source code of Opentaps in the method `invoiceSuppliesOrWorkEffortOrderItems` of the class `InvoiceServices` in line 2276.

```
<target name="create-admin-user-login"
    description="Prompts␣for␣a␣user␣name,␣then␣creates␣a␣user␣with␣admin␣privileges␣and
␣␣␣␣a␣temporary␣password␣equal␣to␣'ofbiz';␣after␣a␣succesful␣login␣the␣user␣will␣be
␣␣␣␣prompted␣for␣a␣new␣password.">
    <input addproperty="userLoginId" message="Enter␣user␣name␣(log␣in␣with
␣␣␣␣the␣temporary␣password␣'ofbiz'):"/>
    <antcall target="load-admin-user-login"/>
</target>
```

Figure 2: A motivating example that shows how default login information is left in a configuration build artifact `build.xml`.

unique; to the best of our knowledge, there exists no prior approach that addresses the problem that we pose in this paper. In summary, we make the following main contributions:

- We create a new data privacy and program comprehension framework (described in Section 4) that includes a novel combination of an entropy-based privacy metric with program comprehension metrics to enable organizations to determine how to balance the goal of redacting sensitive information in software artifacts with a goal of preserving program comprehension.
- We design and implement an algorithm using associative rule mining for determining candidate words that should replace sensitive words when redacting software artifacts.
- We combine our privacy framework with this algorithm and a sound renaming algorithm in RESIST to automatically redact sensitive words in software artifacts.
- We evaluated our data privacy and program comprehension framework experimentally using 57 professional programmers from over a dozen different organizations. Our evaluation shows that RESIST effectively redacts software artifacts thereby making it difficult for participants to infer sensitive information while maintaining program comprehension at a desired level.
- We evaluated performance of RESIST using three open-source Java applications and one commercial Java application ranging from 10 kLOC to 500 kLOC. We show that RESIST is lightweight and can be used for redacting sensitive information in large-scale software.

2. A MOTIVATING EXAMPLE

We use a motivating example from *Opentaps*[4], a large-scale open source integrated application suite that combines customer relationship management, warehouse and inventory management, supply chain management, and financial management with business intelligence and mobility integration for different enterprises. Opentaps is typical of enterprise-level commercial applications, and since we cannot use motivating examples in this paper that involve sensitive information from internal commercial applications, Opentaps provides a close equivalent of examples that we observed in different commercial applications. That is, in this paper, we treat Opentaps as if it was a proprietary commercial application. Opentaps is distributed along with its business documentation that is available from the website[5].

Many companies consider any information as sensitive if it is related to trade secrets and disclosing it may help competitors to gain cost-saving advantage. An example of sensitive information is that a company outsources manufacturing of some products or components to external vendors [8, 32]. Opentaps documentation contains a section that describes a procedure for outsourcing manufacturing tasks[6]. Sensitive information about outsourcing manufacturing can be redacted in this documentation; however, the question is if it is possible to infer this sensitive information from the source code of Opentaps.

Consider a motivating example that is shown in Figure 1. As it turns out, programmes encoded sensitive information about outsourcing manufacturing in the source code of Opentaps in the method `invoiceSuppliesOrWorkEffortOrderItems` of the class `InvoiceServices` in lines `2276-2277`. These two lines of the source code contain plenty of clues for an attacker to infer the sensitive information that the company that owns this application outsources manufacturing. First, the word "`outsourced`" is present in the comment. Second, words "`work effort`" are highly correlated with outsourcing both in the documentation and on the Web, especially when used in conjunction with the word "`invoicing`." The identifier `workEffortIdCompleted` has a very descriptive name that reveals a part of the implemented requirement, and this identifier is defined and used in the source code in different contexts, which contain additional clues that reveal sensitive information about outsourcing manufacturing.

The other example shows how sensitive information is left in software artifacts that are not the source code. An Opentaps document on security controls contains sensitive information for system administrators on passwords and login names[7]. Suppose that a business analyst redacts this documentation to remove the content of the section that contains default login names and passwords. However, this is not enough – a simple attack can reveal login name and password from a software artifact.

[4]http://www.opentaps.org, last checked March 19, 2013

[5]http://www.opentaps.org/docs/index.php/Main_Page

[6]http://www.opentaps.org/docs/index.php/Outsourcing_Manufacturing

[7]http://www.opentaps.org/docs/index.php/Introduction_to_opentaps_Security_Controls

This attack can be carried out as follows. First, an attacker searches for common administrator login names on the Web. The first top five results from Google reveals that the name "`admin`" is common for different applications. After searching the source code for the word "`admin`" the attacker find a build configuration file called `build.xml` in the root of the project tree. This file contains complete login information (i.e., user name is `admin` and the password is `ofbiz`), and the fragment of this file is shown in Figure 2. A lesson from these motivating examples is that it is not enough to redact sensitive information from business and requirements documents, since source code and software artifacts contains this sensitive information. These motivating examples are quite representative of the widespread privacy problems, as we confirmed it during interviews with security experts at Accenture and a major bank. Attackers can easily infer sensitive information from software artifacts, and there is no approach that enables business analysts to address the problems of redacting sensitive information in these artifacts while balancing program comprehension.

3. THE RESIST APPROACH

In this section we explain the problem of balancing utility and privacy, present the gist our approach, describe the architecture of RESIST, and provide an overview of how RESIST is used.

3.1 The Attacker Model

Attacker models in privacy are fundamentally different from attacker models in secrecy. The latter involves a sender who transmits a message to a recipient, and an attacker attempts to block, impersonate, or tamper with the message to change its content to achieve some malicious goal. In this model for secrecy, the sender must share messages with the recipient and cryptographic protocols that usually use encryption are used to protect these messages from the attacker. Thus, a fundamental property of the model for secrecy is sharing messages between the sender and the recipient.

Contrary to this model, a fundamental property of the model for privacy is that *sensitive information* cannot be shared at all. It should be impossible to infer sensitive information about a participant or an entity in a privacy model. In this respect, everyone is an attacker from the perspective of the owner of sensitive information, even if these attackers do not intend to use sensitive information in any malicious way. Moreover, the notion of sensitive information is subjective – what is considered sensitive for one "attacker" may not be interesting at all for some other "attacker." Because of this subjectivity, it is difficult for stakeholders to properly redact documents, not to mention the source code of applications. This subjectivity also makes it difficult to evaluate approaches that redact sensitive information in documents.

3.2 Balancing Utility And Privacy

Utility of anonymized data is measured in terms of usefulness of this data for computational tasks when compared with how useful the original data is for the same tasks. Consider an example of the utility of calculating average salaries of employees. Adding random noise to protect salary information will most likely result in computing incorrect values of average salaries, thereby destroying utility of this computation. Recent cases with U.S. Census show that applying data privacy leads to incorrect results [2, 6]. Multiple studies demonstrate that even modest privacy gains require almost complete destruction of the data-mining utility [1, 10, 24, 25, 37].

We consider program comprehension as a utility of software artifacts, which can be reduced if sensitive words are replaced. Replacing all sensitive words with blanks or randomly generated characters increases the privacy of software artifacts, however, it destroys the utility of program comprehension. Our goal is to enable stakeholders to balance privacy and utility goals, i.e., to choose appropriate levels of privacy that will guarantee certain level of program comprehension.

Consider a graph where the horizontal axis shows the values of privacy, P, and the vertical axis shows the values of program comprehension, C. We normalize these values to lie between zero and one. For $P = 0, C = 1$ meaning that if no sensitive words are replaced in software artifacts, the level of privacy is zero and the level of program comprehension is the same as it was before, we say that the highest and equal to one. Conversely, $P = 1, C \to 0$ means that if all sensitive words are replaced with blank or random characters, the level of privacy is one meaning that we achieved full privacy and the level of program comprehension is reaching zero in the limit. We call this graph a *PC–graph*, and computing this graph enables stakeholders to balance privacy and utility goals, i.e., to choose appropriate levels of privacy that will guarantee certain levels of program comprehension.

We need intermediate levels of privacy. Most existing privacy approaches achieve full privacy w.r.t. some privacy metric, and it is considered too inflexible. Guessing anonymity and computational differential privacy are examples of some approaches that achieve intermediate results where some privacy is lost to achieve better utility. This is the approach that we take in this paper.

3.3 The Gist of RESIST

The gist of our approach is twofold. First, RESIST determines automatically how to find words in software artifacts that may enable attackers to infer sensitive information. Specifically, RESIST does not specify what sensitive information is – in fact, only business analysts and security experts can decide what constitutes sensitive information. That is, business analysts and security experts redact sensitive information from business and requirements documents by specifying sensitive words. Once this redaction is completed, RESIST uses these sensitive words and other words that co-occur with sensitive words in given contexts to infer other related words that will help attackers infer sensitive information. That is, in the motivating example, RESIST will determine that the words "`work effort`" and "`invoice`" are related to outsourcing manufacturing.

Second, RESIST finds automatically a list of candidate words that can replace sensitive words. To do that, it can use business documents and the Web, and in our experiments we used Wordnet[8]. For each word RESIST learns an associative rule that shows with what level of confidence a candidate word can replace some sensitive word. In addition, RESIST determines how program comprehension will change for the application if candidate words replace sensitive words. RESIST will also use its privacy metric to determine how well sensitive information is protected if certain candidate words are chosen for replacement. In the end, it is helping analysis to choose the right balance between the privacy level and program comprehension.

To illustrate the second part of RESIST, consider a situation where sets of words S_1 ={"`transaction`", "`bill`"} and S_2 = {"`task`", "`report`"} are considered as candidates to replace sensitive words "`work effort`" and "`invoice`" in our motivating example. These sets of words are found by analyzing co-occurrences and their frequencies with the sensitive words on the Web. Words from the set S_1 co-occur with the sensitive words much more frequently in the context of outsourcing manufacturing than the words from the set S_2, making it easier for the attacker to

[8]http://wordnet.princeton.edu

infer sensitive information. However, using the words from the set S_2 makes it more difficult for personnel to relate the code that is shown in Figure 1 to the specific requirement, thereby complicating tasks for which this personnel is hired. The ultimate decision is made by business analysts and security experts who determine a proper balance.

3.4 RESIST Architecture And Workflow

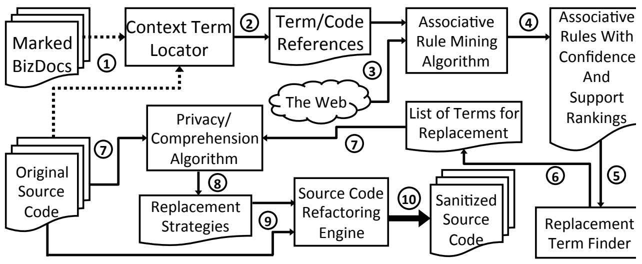
The architecture of RESIST is shown in Figure 3. Arrows show command and data flows between components, and numbers in circles indicate the sequence of operations in the workflow. The beginning of the workflow is shown with the dotted arrows labeled (1) that indicates that input to RESIST is the source code and the set of business documents and requirements with sensitive words marked by business analysts and security experts. The workflow end is shown with the fat solid arrow labeled (10) that indicates the output of RESIST is the sanitized source code.

In step (1), the Context Term locator analyzes the source code and the set of business documents and requirements with marked sensitive words (also called terms), and it outputs (2) term/code references that specify terms that co-occur with sensitive words in the same contexts and their locations in the source code and other software artifacts. In this paper, we call context a unit of cohesive information, which we choose to coincide with a paragraph.

The next step is to determine a list of candidate words that can replace sensitive words and words that frequently co-occur with sensitive words. To do that, we approximate the universal word reference corpus with the Web, which has been successfully used to represent the adversary's knowledge [15, 53]. To determine replacement words, we obtain association rules that in general describe relations between different elements as implication expressions $A_1 \wedge A_2 \wedge \ldots \wedge A_i \Rightarrow C_1 \vee C_2 \vee \ldots \vee C_j$, were A and C are disjoint itemsets, that is $A \cap C = \emptyset$ [54, pages 327–332]. There are different algorithms for extracting association rules from documents, and the quality metrics for extracted rules are support and confidence. Support measures the applicability of an association rule to a dataset, and confidence estimates the frequency with which items in C also appear in query results that contain A. Support, s, and confidence, c are calculated using the following formulae: $s(A \Rightarrow C) = \frac{\sigma(A \cup C)}{N}$ and $c(A \Rightarrow C) = \frac{\sigma(A \cup C)}{\sigma(A)}$, where σ is the support count and N is the total number of results.

Then, both term/code references and the Web are the input (3) to the Associative Rule Mining Algorithm that outputs (4) the set of associative rules with confidence and support rankings. These rules describe candidate words that can be used to replace sensitive words from term/code references. In addition, if these rules show that a sensitive word is uniquely defined by some candidate word, then this candidate word is added to the list of sensitive words. Searching for replacement words is done using either the Web or a local collection of documents. A study was carried out for learning association rules both using the Web and local collections of documents[15], and we extend this previous work in RESIST.

Associative rules are analyzed (5) by the Replacement Term Finder that outputs (6) the List of candidate Terms for Replacement of sensitive words. Given that many associative rules can be obtained in step (5), it is necessary to select a subset of them that offer viable alternatives for sensitive words. The selection is dictated by the level of privacy as well as the need to retain a desired level of program comprehension. The list of candidate term replacements is inputted (7) to the Privacy/Comprehension Algorithm along with the original source code, and this algorithm determines (8) Replacement Strategies, which is the ordered list of terms that should replace words or parts of identifiers in the source


Marked BizDocs
1
Context Term Locator
2
Term/Code References
Associative Rule Mining Algorithm
4
Associative Rules With Confidence And Support Rankings
3
The Web
List of Terms for Replacement
Privacy/ Comprehension Algorithm
7
7
Original Source Code
8
6
5
Replacement Strategies
9
Source Code Refactoring Engine
10
Sanitized Source Code
Replacement Term Finder
Figure 3: The architecture and the workflow for RESIST.

code. Sometime, it may not be possible to use a given term for replacement due to naming conflicts or constraints of the given language. This is why different alternatives should be given (9) as the input to the Source Code Refactoring Engine that makes actual replacements.

We select a name replacement algorithm that lead to sound binding of names to declarations in Java [49]. In general, RESIST can work with applications that are written in different languages as long as sound renaming approaches can be plugged into the RESIST architecture [43, 57, 28]. The refactoring engine outputs (10) Sanitized Source Code, and this step completes the workflow of RESIST.

3.5 Using RESIST

RESIST can be used in two modes: the batch mode for computing a *PC*–graph, and the interactive mode for users to sanitize software artifacts by replacing selected sensitive words. A goal of the batch mode is to determine the distribution of *PC* values for different replacements of different sensitive words. Knowing this distribution is important for stakeholders who need to understand how redacting documents and software artifacts will affect the balance between privacy and program comprehension. The batch mode is fully automatic, where sets of sensitive words are selected at random from documents, and for each set RESIST computes the *PC* values that are put as a dot on the *PC*–graph. As a result, stakeholders obtain a quantitative way to reason about the effect of choosing different words as sensitive on the values of privacy and comprehension.

The interactive mode enables stakeholders to redact software artifacts by studying the effects of sanitizing these artifacts by replacing sensitive words. Initially, stakeholders determine the ranges of values for privacy and program comprehension that they want to achieve. After they produce the *PC*–graph, the stakeholders can zoom in into the region of the graph with the desired ranges of privacy and comprehension values. The zoomed-in region contains a number of points that are computed for different sensitive words and their replacement values. Stakeholders review these sensitive words and determine which ones should be redacted and how they affect program comprehension and privacy values. The interactive mode also allows stakeholders to select different replacement words and study their effect on the *PC* metric. Once the final selection of sensitive words and their replacement words is done, RESIST will sanitize software artifacts automatically by refactoring source code of these artifacts.

4. THE FRAMEWORK

In this section, we discuss the *privacy and comprehension metrics (PCM)* as a single framework for quantifying the amount of sensitive information that is released to third parties when application's source code is revealed.

4.1 Constraints and Goals

A purpose of this paper is to offer a joint *privacy and comprehension metrics (PCM)* in a single framework for quantifying the amount of sensitive information that is released to third parties when application's source code is revealed. Several constraints affect the choice of PCM: simplicity, language-neutrality, and flexibility to be used at different levels of module granularity. Specifically, computing PCM should not require too much time or a significant amount of resources, and this objective can be achieved by simplifying the logic and subsequently the computation of the PCM. In addition, PCM should scale to handle a large number of sensitive as well as non-sensitive words from the application's source code.

Language neutrality is achieved by dealing with plain text words that are extracted from the application's source code, that is the source code is preprocessed to extract the names of identifiers and words from comments. Finally, the flexibility to deal with different granularities of programming modules enables variations of comprehension metric algorithms, which are sensitive to the granularities of the modules and dependencies among them. These constraints dictate the design of the proposed PCM and make it difficult to select off-the-shelf anonymization algorithms that may violate some of these constraints.

4.2 The Privacy Metric

In this section, we describe the privacy metric that we developed for RESIST.

4.2.1 Privacy As A Model of Attacker

Recall from Section 3.1 that from the perspective of the owner, everyone else is an attacker who try to guess sensitive information from sanitized documents. Therefore, to measure privacy of a document means to measure the difficulty that it takes for an attacker to guess sensitive information given the information in sanitized documents. That is, the attacker attempts to reverse engineer sensitive word replacements using different techniques, including but not limited to pure guessing strategies based on available non-redacted information in documents. The difficulty that the attacker experiences when reverse engineering sensitive information is directly proportional to the closeness or support for replacement words with respect to their corresponding sensitive words. That is, reverse engineering a sensitive word from a random replacement string is much more difficult when compared to doing it from a semantically close replacement word (e.g., reverse engineering the sensitive word `HIV` from the replacement word `antiretroviral`). Thus, a privacy measure should reflect this difficulty and this measure could be normalized between zero and one, that is, between original sensitive words remaining in the document to complete random replacements.

4.2.2 Entropy Is A Measure of Privacy

Suppose that the set of events, $E = (e_1, \ldots, e_n)$ has the set of probabilities associated with each event, $P = (p_1, \ldots, p_n)$. Shannon's definition of the entropy is $E(P) = -\sum_{i=1}^{n} p_i \cdot \log p_i$. Entropy is equated with the average amount of information of some random process, which in our case is replacing sensitive words with replacement words whose support is used as the probability of guessing the sensitive words by analyzing their respective replacement words. The probabilities are normalized over the event space, so that their sum is equal to one.

Entropy-based privacy loss is a well-known metric that is used in statistical databases [59, pages 76–85]. The main idea for this metric is that when some statistical information is released about data in a database (e.g., the average salary of employees), the entropy of each record in this database changes, since certain information is released. Suppose that $E(R)$ is the entropy of a record in the database prior to the release of the information, and $E'(R)$ is the entropy of a record in the database after the release of the information. Then the privacy loss, $\triangle PL$ is computed as $\triangle PL = \frac{E(R)-E'(R)}{E(R)}$. We can compute $PM = 1 - \triangle PL$.

We chose an *entropy-based privacy metric (EPM)* for a number of reasons [59, page 76]. EPM can be universally applied to any statistical disclosure and it does not depend on specific data types, or formats of data structures, or languages in which this data is defined. More importantly, EPM enables easy comparability of different transformational techniques applied to documents such as code refactoring that we use in this paper. Easy comparability of different EPMs is important, since RESIST is extensible to incorporating and evaluating other techniques in the future, thereby allowing researchers to determine different trade-offs between different techniques and measures.

4.2.3 Entropy-Based PM For RESIST

In order to compute PM for RESIST, the entropy $E(M)$ should be computed respectively for the source code module, M, where sensitive words are replaced with words (replacement words) to protect privacy. Each module contains the collection of sensitive words, $S = \{s_1, \ldots, s_n\}$, and non-sensitive words, $C = \{c_1, \ldots, c_m\}$. For each sensitive word, s_k, there is a set of replacement words, $R = \{(r_1, P_{r_1}), \ldots, (r_q, P_{r_q})\}$, where r_k is a replacement word and P_{r_k} is the support calculated for this replacement word and its corresponding sensitive word, s_k.

The minimum entropy $E_{min}(M)$ shows the level of information that is contained in the original source code module, M. It is measured by quantifying the relationship between sensitive and non-sensitive words in this module. That is, each sensitive word, s_k is related to each of the non-sensitive words in C, and this relationship is measured by the support between sensitive and non-sensitive terms. For example, if the word "`HIV`" is the sensitive word that is used to name an identifier, removing it will not change the fact that the information about HIV can be uniquely identified by the non-sensitive word "`Prezista`," which is a drug used in HIV treatment. To compute the entropy $E_{min}(M)$, we first obtain the matrix $V_{min} = \|C \times S\|$ whose cells contain normalized confidence values between each non-sensitive and sensitive terms as the probabilities whose sum is equal to one. Using this matrix, $E_{min}(M) = -\sum_{j=1}^{m} \sum_{i=1}^{n} V_{ji} \cdot \log V_{ji}$.

The maximum entropy, $E_{max}(M)$, can be calculated in documents where all sensitive words are replaced with random strings, that is the information loss is maximum. For computing $E_{max}(M)$ we use the same formula on the matrix $V_{max} = \|C \times G\|$, where cells hold confidence values between non-sensitive terms and random string replacements, G. The difference between $E_{max}(M) - E_{min}(M)$ specifies the range of entropy values that the document can have after sensitive words are replaced with replacement words.

Once a set of replacement words are selected, that is $RS = (s_l, r_t)$, $\ldots, (s_u, r_v)$, and the matrix $V = \|C \cup S \times RS\|$, where cells hold confidence values between all terms and replacement words. we compute $E'(M) = -\sum_{i=1}^{|RS|} P_{r_k} \cdot \log P_{r_k}$. Finally, $E(M) = \frac{E'(M)-E_{min}}{E_{max}(M)-E_{min}(M)}$.

4.3 The Comprehension Metrics

Early work on program comprehension [44, 58] highlighted the significance of textual information to capture and encode the knowledge of the software developers. Identifiers used by programmers as names for classes, methods, or attributes in source code or other

Table 1: Characteristics of the subject applications and their documents. App = application code size(.java files), Docs = size of documentation, WD = total words in documents, WJ = total words in java files, MW = words in docs that match words in source, CW = total words from documentation found in Java comments, SI = total words from documentation found concatenated in identifier names.

Subject Application	App [LOC]	Docs [MB]	WD	WJ	MW	CW	SI
The Cobalt Group PersonalPages	9,502	0.226	17,191	29,070	767	627	614
Onebook	15,775	0.21	31,304	71,570	1,466	1,343	NA
ImageJ	97,892	12	853,733	366,856	11,442	6,279	7,782
Opentaps	500,603	82	5,194,037	2,259,586	10,980	8,500	NA

artifacts contain important information [39, 3] and account approximately for more than a half of the source code in software [23]. These names are often used in solutions to many program comprehension tasks [12], since these names reflect the concepts that programmers encode [3]. Capturing conceptual relations between names of different identifiers is the essence of the comprehension metric that we use in this paper.

In this paper, we use *Conceptual Cohesion of Classes (C*3*)* and *Conceptual Coupling of a Class (CoCC)*, which capture the conceptual aspects of class cohesion and coupling, as they measure how strongly the methods of classes relate to each other conceptually. The conceptual relation between methods is based on the principle of textual coherence in cognitive psychology and computational linguistics [38], and we utilize this principle in these comprehension metrics that have been successfully used and evaluated by Poshyvanyk and Marcus [40, 45]. Also, the conceptual metrics have been previously evaluated in a user study with software developers [47]. The results indicate that the conceptual coupling metrics at the feature level align with developers' opinions of feature coupling. We do not include complete definitions and formulas for *C*3 and *CoCC* metrics for space limitation reasons, however, the complete details behind the metrics, necessary to reproduce the results based on these measures can be obtained from our preliminary work [40, 45].

*C*3 and *CoCC* are based on the analysis of textual information in source code, expressed in comments and identifiers. *Latent Semantic Indexing (LSI)* is used to analyze the textual information from source code [22]. *C*3 and *CoCC* can be interpreted as measures of the textual coherence of classes within the context of the entire system, and cohesion and coupling ultimately affects comprehensibility of source code. That is, to easily understand source code, it has to have a clear implementation logic (i.e., design) and it has to be easy to read (i.e., good and consistent use of identifiers). These two properties are captured by conceptual cohesion and coupling metrics, respectively.

The nature of the metrics *C*3 and *CoCC* is that they provide insight into the change in conceptual relationships between program entities before and after redaction of sensitive information in source code. When replacing sensitive terms the change is induced in conceptual relationships between source code entities that serves as an indicator of change in program comprehension. For instance, consider a software system which contains the methods `LoadHIVPatientList` and `SaveHIVPatientList`. In this example, the term "`HIV`" is sensitive and all occurrences are replaced after redacting sensitive words in the source code. Thus, replacements could produce the names `LoadSickPatentList` and `SaveAilingPatentList`, where "`HIV`" is replaced by "`Sick`" for one occurrence and "`Ailing`" for the other. The relationship between the methods `Load` and `Save` may not be obvious to stakeholders anymore. As a result, stakeholders spend additional effort to identify the relationship between these two methods when comprehending the source code. For the specified example, *C*3 decreases as a result of modifying the source code because of the lack of consistency in using terms. The conceptual metrics provide a mechanism to capture these conceptual changes between methods and classes based on the analysis of identifiers and comments in the source code.

A goal of RESIST is to protect and hide sensitive information in source code. Sensitive words are identified and replaced with alternative words. In doing so, it is possible that performed replacements may impact program comprehension in underlying modules. This is because altering terms in the source code may impact conceptual relationships which developers use during program comprehension activities.

5. EXPERIMENTAL EVALUATION

In this section, we pose research questions (RQs), describe the methodology for experimental evaluation to answer these RQs and subject applications, discuss threats to validity, and analyze the results of this evaluation.

5.1 Research Questions

In this paper, we seek to answer the following research questions.

RQ1: How does redaction of sensitive information impact program comprehension? More specifically, does redacting sensitive information in software artifacts make it more difficult to correctly guess sensitive information from sanitized artifacts?

RQ2: How effective is RESIST in computing different levels of data privacy for different levels of program comprehension?

With RQ1, we seek to answer the observational/relational question that determining how levels of privacy affect program comprehension. We hypothesize in this paper that achieving higher privacy negatively affects program comprehension, however, we need to test this hypothesis on subject applications. This RQ is important to show that with RESIST, the privacy metric is linked directly to program comprehension and vice versa; in other words, guaranteeing a certain level of program comprehension should allow stakeholders to calculate bounds of the privacy level.

With RQ2, we address our claim that using different levels of privacy enables stakeholders to make trade-off decisions about privacy and utility in acceptable period of time with the acceptable use of resources. Since RESIST helps stakeholders to make business decisions, the tool should be lightweight and not resource-demanding, making it effective for this task without requiring significant investment.

5.2 Experimental Methodology For RQ1

We used a standard experimental design in a cohort of 57 participants who were randomly divided into control and treatment groups. The experiment was carried out using the Internet[9]. The control group was given original software artifacts and the treatment group was assigned redacted Java source code files from different projects, specifically, from the released source code of the tool PCAnywhere by Symantec, from Mozilla, and from Opentaps. The authors of this paper selected sensitive words in the selected files and formulated questions, answers to which required understanding of the source code and knowing the meaning of the sensitive words. Replacement words were chosen automatically and the code was refactored accordingly using RESIST. Below are example questions that the participants were asked to answer after studying the original or redacted source code.

1. Is getting permission for access to swap files linked to obtaining privileges for a cache file?

2. This file is from the source code of a product by Symantec. Is there evidence of external collaboration with Novell?

3. Do we have any evidence of ThinHost logon failing for lower case usernames?

The fact that the authors of this paper selected sensitive words does not imply bias in the experimental methodology. Consider the fact that identifying sensitive information is subjective, and any word besides obvious common English words and articles (e.g., the, get, or go) can serve as a sensitive word. The notion of a "sensitive" word is highly subjective, and only business analysts and security experts can make decisions what words are sensitive for given domains. Therefore, any stakeholder can identify sensitive words based on business or security rationale. What we evaluate in this experiment is how RESIST redacts sensitive information in software artifacts, and RESIST accomplishes redaction without any interference from the authors of this paper.

In the course of the experiment, Java source code files were redacted to achieve the level of privacy 0.5. Different comprehension levels were achieved for different replacements for different files. Participants answered questions based on their understanding of the source code of the files, and each participant accomplished this step individually, assigning a confidence level, *C*, to the examined code using the five-level Likert scale from the following guidelines.

1. I can easily find the sensitive information in the software artifact. The score for this answer is four.

2. I can infer the sensitive information from the software artifact with some effort. The score for this answer is three.

3. I am likelier to infer from the software artifact the opposite of the statement that defines sensitive information. The score for this answer is two.

4. I can infer from the software artifact the opposite of the statement that defines sensitive information. The score for this answer is one.

5. I cannot infer the sensitive information from the software artifact at all. The score for this answer is zero.

[9]http://www.cs.wm.edu/semeru/resist

Fifty seven participants are from a dozen of different organizations that include six graduate students from the University of Illinois at Chicago, seven graduate students from the College of William and Mary, eight Accenture employees who work on consulting engagements as professional programmers for different client companies, and the rest of participants are professional programmers from different software companies. All participants have at least one year of Java experience. Participants have different backgrounds, experience, and belong to different companies that build software for different domains. A majority of participants are members of the ACM SigSoft group on Linkedin.

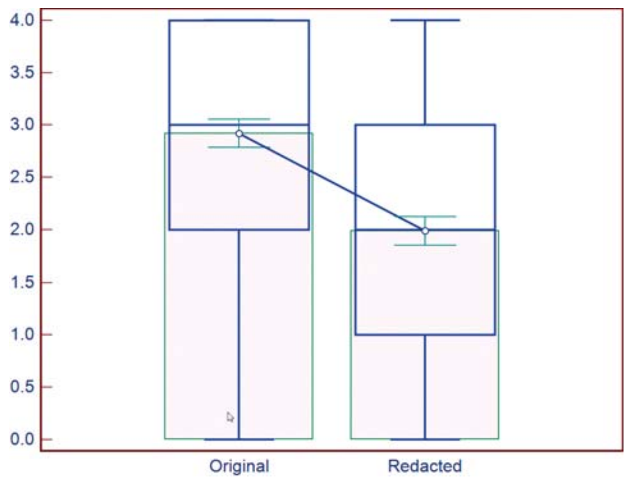


Figure 4: Statistical summary of the results of the experiment for the confidence level, C, of the participants who identified sensitive information in software artifacts, original and redacted.The central box represents the values from the lower to upper quartile (25 to 75 percentile). The middle line represents the median. The thicker vertical line extends from the minimum to the maximum value. The filled-out box represents the values from the minimum to the mean, and the thinner vertical line extends from the quarter below the mean to the quarter above the mean.

5.3 Subject Applications

We evaluate the performance of RESIST for answering RQ2 with three open-source and one commercial Java applications that belong to different domains. Our selection of subject applications is influenced by several factors: size of the documentation, size of the source code, popularity of the application, how an application is representative of other applications.

`Opentaps` is an application that we use as motivating example throughout this paper. `Onebook` is a Web-based application which allows students and teachers to share information.[10]. `Personal-Pages` is a commercial application that enables auto dealerships to manage customer data. Finally, `ImageJ` is a Java image processing and analysis application.[11]

Table 1 contains characteristics of the subject programs, with the first column showing the names followed by other columns with different characteristics of these applications as specified in the caption. The source code of the project ranges from less than 10 kLOC to over 500 kLOC. Documentation for these subject appli-

[10]http://onebook.sourceforge.net as of Sept 20, 2011.
[11]http://rsbweb.nih.gov/ij/index.html as of Sept 20, 2011.

Table 2: Results of the experiment with participants for original (O) and redacted (R) documents with the privacy value 0.5. Avg-O stands for the average results for the Original artifacts Avg-R stands for Redacted artifacts. % R/O shows the percentage difference between the corresponding values for Redacted and Original artifacts. The last column shows the combined values of cohesion and coupling for program comprehension.

	Survey Responses			Cohesion			Coupling			Comprehension		
File	**Avg-O**	**Avg-R**	**% R/O**	**O**	**R**	**% R/O**	**O**	**R**	**% R/O**	**O**	**R**	**% R/O**
NetscapeSecurity.java	2.32	1.50	-35.4%	1.00	1.00	0.0%	0.79	0.78	-1.2%	0.90	0.89	-0.5%
RegistryUserManager.java	2.53	1.85	-26.8%	0.26	0.22	-13.3%	0.81	0.76	-6.2%	0.53	0.49	-7.9%
Secret.java	3.13	2.13	-31.9%	0.27	0.22	-17.7%	0.80	0.78	-2.6%	0.53	0.50	-6.4%
NetworkIO.java	2.57	1.15	-55.3%	0.16	0.16	-0.2%	0.77	0.72	-7.0%	0.47	0.44	-5.8%
SecurityDialog.java	3.43	2.65	-22.8%	0.12	0.12	0.0%	0.79	0.74	-6.3%	0.45	0.43	-5.5%
GWPassword.java	3.20	3.18	-0.6%	0.20	0.20	0.0%	0.80	0.78	-3.2%	0.50	0.49	-2.6%

cations came from different sources including manuals and original requirements that are available along with the source code of these subject applications.

5.3.1 Variables

Two main independent variables are selecting sensitive words and selecting replacement words. The latter has a confounded variable, i.e., confidence of the replacement word with respect to its corresponding sensitive word. Since RQ1 is to determine how program comprehension depends on privacy, selecting replacement values with lower confidence values should lead to lower levels of program comprehension, which is our hypothesis. A dependent variable is the confidence level, C. We report these variables in this section. The effects of other variables (question description length, prior knowledge) are minimized by the design of this experiment.

The main response variables are the values of privacy and program comprehension for *PC*–graphs, on which we perform trend analysis for RQ2. Additional response variables are time and memory consumption that it takes to computer *PC*–graph to answer RQ2.

5.3.2 The Structure of the Experiment for RQ2

To evaluate the performance of RESIST for answering RQ2, we select statistically representative samples as subsets of sensitive words by randomly choosing them from the sets of all words. For the experiments, we select as potential sensitive words all non-stop words in documents, giving preference to those words that have matches in software artifacts of the corresponding subject applications. It is physically not possible to carry out an experiment using all possible subsets of the powerset of words in documents as sensitive words. Given that documents of the subject applications contain tens of thousands of words, it is challenging to select a large number of subsets of them as sensitive words, so in this experiment we randomly sampled the entire space of words. Our goal is to run experiments for different sensitive words and for different replacement words for these sensitive words with different values of confidence, and report the effects these values on the dependent variables.

5.4 Threats to Validity

A threat to the validity of this experimental evaluation is that our subject programs are of small to moderate size because it is difficult to find a large number of open-source programs that contain sufficient documentation. Large applications that have millions of lines of code and have different documents whose sizes are measured in hundreds of thousands of words may have different characteristics compared to our small to medium size subject applications. Increasing the size of applications to millions of lines of code may lead to a nonlinear increase in the analysis time and space for RESIST. Future work could focus on making RESIST more scalable.

Another threat to validity is that for open-source applications, documents are mostly user manuals, and it is likely that these documents were created after the source code of subject applications was written. It may mean that programmers did not encode sensitive words in software artifacts, however, it should not affect the results of our experiments for two reasons. First, the notion of a sensitive word is highly subjective and a word that is considered sensitive by one stakeholder at one time may not be considered sensitive by a different stakeholder or even by the same stakeholder at a different time, since the notion of trade secret, for example, is time-sensitive. Second, we also use a commercial application in our evaluation with documents that guided programmers in writing software artifacts, and after qualitatively comparing results from open-source and this commercial applications, we see that they are fundamentally the same.

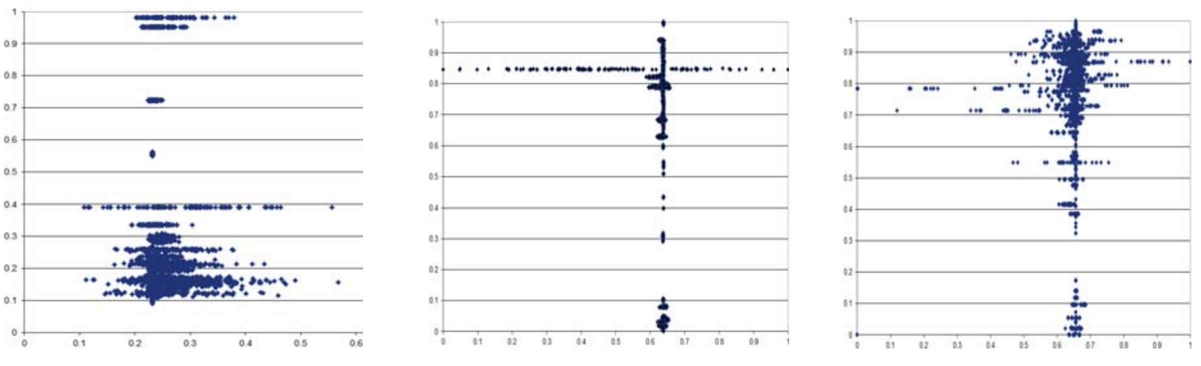
5.5 Results

The results of the experiment to answer RQ1 are shown in Table 2. We carry out experiments with the participants for answering RQ1 using different Java source code files from different projects, some of which were leaked on the Internet. The summary graph for the levels of confidence is shown in Figure 4, and after applying statistical t-tests and z-tests to compare confidence values, we obtained statistically significant difference $(p < 0.05)$ that participants showed better comprehension for original software when compared with redacted one.

The results of the experiments conducted to address *RQ*2 are shown in Figure 5 as *PC*–graphs for three subject applications out of four due to space limitation. The *PC*–graphs for all subject applications exhibit the same pattern, where there is a vertical line composed of different dots at some privacy value, and different horisontal lines of different lengths for different P levels. It is surprising that the same pattern emerged for all of the applications.

We explain the horizontal lines in the pattern as a result of implementation of different requirement topics. For example, replacing some sensitive words may destroy one topic while preserving others, thereby making the application sensitive to joint distributions of sensitive words that may be destroyed by using replacement words. We also observe that the contents of the *PC*–graphs show both gradual distributions of dots in some regions as well as tightly clustered groupings. This variety makes it easier for stakeholders to balance privacy and program comprehension on a continuous scale.

We ran full experiments to select 16 sensitive words (a number of sensitive words was chosen experimentally), find a list of ten alternative words for each sensitive word, replace these words with



(a) Personal Pages. (b) Onebook. (c) ImageJ.

Figure 5: Experimental results for three subject applications. **All *PC*–graphs show the dependence on the privacy level that is assigned to the horizontal axis. The vertical axis designates program comprehension metric.**

combinations of alternative words, and compute *P* and *C* values. Experiments were carried out on Intel Xeon CPU E5620, 2.40GHz and 1.3Gb RAM CentOS release 5.7. The elapsed execution time ranges between 102 minutes to 626 minutes for selecting sensitive words; from 33 minutes to 3,481 minutes for find alternative word replacements. Total elapsed execution time for applying RESIST to compute *PC*–graphs for four subject applications for 100 points for each graph ranges from 436 minutes for Personal Pages to 4,290 minutes for Opentaps.

These results are encouraging especially considering the limited resources that we allocated, since RESIST's replacement word finder slowed down execution and increased memory consumption when using the Web as a source of replacement words. This is significant, but we expect future versions of the RESIST implementation to reduce this overhead by caching data from the Web resources.

Result summary. These results suggest that RESIST helps stakeholders to effectively quantify the trade-off between program comprehension and privacy when redacting sensitive information in software artifacts for subject applications, thereby addressing RQ1. The results also suggest that RESIST is effective in helping stakeholders balance program comprehension and privacy since it is lightweight and does not require significant resources, thereby addressing RQ2.

Recall that selecting sensitive words is a subjective process – what is sensitive for one individual or organization may not be sensitive to another. Since ours is the first approach that addresses a new problem of redacting sensitive information in software artifacts, our goal is to conduct a qualitative evaluation of determining if this approach leaves sufficient choices to stakeholders. In this paper, we focus on an exploratory case study in which we evaluate the tradeoff of privacy, `P` versus comprehension, `C`. The metric, `C` captures how meaningful the replacements are in that if we insert nonsensical replacements, `C` will be reduced. In our study, we found sets of replacements for which `C` is equivalent to when the original words are used, while `P` is increased. In other words, we can increase privacy without substantially decreasing comprehension (at least for our subject applications). This finding already presents a strong accomplishment and another starting point for future work.

6. RELATED WORK

Our work is related to program comprehension, since RESIST is used to assess the impact of privacy on program comprehension. One of the goals of software design is to create applications whose classes have high cohesion and low coupling. Software cohesion can be defined as a measure of the degree to which elements of a module belong together [7]. Proposals of measures and metrics for cohesion and coupling are abound in literature, as these software metrics proved useful in different tasks [19] including, but not limited to, assessment of design quality [5, 9], productivity, design and reuse effort [14], modularization of software [39], and identification of reusable components [27]. However, no work addressed the question of how program comprehension is affected by data privacy.

Related to RESIST are approaches for balancing test coverage and data privacy, specifically, PRIEST [55] and TADA [30] as well as $kb-$anonymity model that enables stakeholders to release private data for testing and debugging by combining the $k-$anonymity with the concept of program behavior preservation [11]. Unlike RESIST, these approaches are not related to program comprehension, and they replace some information in the original data to ensure privacy preservation so that the replaced data can be released to third-party testers and developers. RESIST and testing-related approaches are related in the idea of using different privacy mechanisms to preserve original data thereby improving its testing utility.

7. CONCLUSION

We created a novel approach for automatically *REdacting Sensitive Information in Software arTifacts (RESIST)* that combines a data privacy metric, associative rule mining for finding sensitive and replacement words, program comprehension metrics, and a sound renaming algorithm for Java programs. We have built a tool for our approach, applied it to nontrivial Java applications, and evaluated it with professional programmers and students. Our results suggest that with RESIST, effective quantification of trade-off is possible between program comprehension and privacy when redacting sensitive information in software artifacts.

8. ACKNOWLEDGMENTS

We warmly thank the anonymous reviewers for their comments and suggestions that helped us in improving the quality of this paper. This material is based upon work supported by the National Science Foundation under Grants No. 1217928, 1017633, 1016868 and Microsoft Research SEIF program. We also thank Xiangli Chen, a graduate student at UIC for participating in the early work on this paper.

9. REFERENCES

[1] C. C. Aggarwal. On k-anonymity and the curse of dimensionality. In *VLDB '05*, pages 901–909. VLDB Endowment, 2005.

[2] J. T. Alexander, M. Davern, and B. Stevenson. Inaccurate age and sex data in the census pums files: Evidence and implications. Working Paper 15703, National Bureau of Economic Research, January 2010.

[3] G. Antoniol, Y.-G. Gueheneuc, E. Merlo, and P. Tonella. Mining the lexicon used by programmers during software evolution. In *ICSM'07*, pages 14–23, Paris, France, 2007. IEEE Computer Society Press.

[4] W. Aspray, F. Mayades, and M. Vardi. *Globalization and Offshoring of Software*. ACM, 2006.

[5] J. Bansiya and C. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE TSE*, 28(1):4–17, 2002.

[6] C. Bialik. Census bureau obscured personal data – too well, some say. *The Wall Street Journal*, Feb. 2010.

[7] J. Bieman and B.-K. Kang. Cohesion and reuse in an object-oriented system. In *ACM SSR'95*, pages 259–262, 1995.

[8] S. M. Bragg. *Outsourcing: A Guide to Selecting the Correct Business Unit, Negotiating the Contract, Maintaining Control of the Process*. John Wiley & Sons, Inc., New York, NY, USA, 2006.

[9] L. C. Briand, J. Wï£¡st, J. W. Daly, and V. D. Porter. Exploring the relationship between design measures and software quality in object-oriented systems. *Journal of System and Software*, 51(3):245–273, 2000.

[10] J. Brickell and V. Shmatikov. The cost of privacy: destruction of data-mining utility in anonymized data publishing. In *KDD '08*, pages 70–78, New York, NY, USA, 2008. ACM.

[11] A. Budi, D. Lo, L. Jiang, and Lucia. *b*-anonymity: a model for anonymized behaviour-preserving test and debugging data. In *PLDI*, pages 447–457, 2011.

[12] C. Caprile and P. Tonella. Nomen est omen: Analyzing the language of function identifiers. In *WCRE'99*, pages 112–122, Atlanta, Georgia, USA, 1999.

[13] C. Casper. Roundup of privacy research, 4q10. *http://www.gartner.com/DisplayDocument?id=1497614*, Dec. 2010.

[14] S. Chidamber, D. Darcy, and C. Kemerer. Managerial use of metrics for object-oriented software: An exploratory analysis. *IEEE TSE*, 24(8):629–639, 1998.

[15] R. Chow, P. Golle, and J. Staddon. Detecting privacy leaks using corpus-based association rules. In *KDD '08*, pages 893–901, New York, NY, USA, 2008. ACM.

[16] L. Constantin. Kaspersky confirms source code leak, threatens legal action against downloaders. *http://news.softpedia.com/news/Kaspersky-Anti-Virus-Source-Code-Leaked-Online-181297.shtml*, Jan. 2011.

[17] T. A. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.

[18] C. M. Cumby. Protecting sensitive topics in text documents with protextor. In *ECML/PKDD (2)*, pages 714–717, 2009.

[19] D. Darcy and C. Kemerer. Oo metrics in practice. *IEEE Software*, 22(6):17–19, 2005.

[20] Datamonitor. Application testing services: global market forecast model. *Datamonitor Research Store*, Aug. 2007.

[21] J. W. Davison, D. Mancl, and W. F. Opdyke. Understanding and addressing the essential costs of evolving systems. *Bell Labs Technical Journal*, 5(2):44–54, 2000.

[22] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41:391–407, 1990.

[23] F. Deissenboeck and M. Pizka. Concise and consistent naming. In *IWPC'05*, pages 97–106, St. Louis, Missouri, USA, 2005.

[24] I. Dinur and K. Nissim. Revealing information while preserving privacy. In *PODS '03*, pages 202–210, New York, NY, USA, 2003. ACM.

[25] J. Domingo-Ferrer and D. Rebollo-Monedero. Measuring risk and utility of anonymized data using information theory. In *EDBT/ICDT '09*, pages 126–130, New York, NY, USA, 2009. ACM.

[26] T. Espiner. Extortion failed - anonymous posts symantec source code. *http://www.shacknews.com/article/28619/half-life-2-source-leak*, Feb. 2012.

[27] L. H. Etzkorn and C. G. Davis. Automatically identifying reusable oo legacy code. *IEEE Computer*, 30(10):66–72, 1997.

[28] A. Garrido and J. Meseguer. Formal specification and verification of java refactorings. In *SCAM'06*, pages 165–174, Washington, DC, USA, 2006. IEEE Computer Society.

[29] S. Gibson. Half-life 2 source leak. *http://www.shacknews.com/article/28619/half-life-2-source-leak*, Oct. 2003.

[30] M. Grechanik, C. Csallner, C. Fu, and Q. Xie. Is data privacy always good for software testing? In *ISSRE*, pages 368–377, 2010.

[31] E. Hull, K. Jackson, and J. Dick. *Requirements Engineering*. SpringerVerlag, 2004.

[32] M. Jesper. Framework for outsourcing manufacturing: strategic and operational implications. *Comput. Ind.*, 49:59–75, September 2002.

[33] T. C. Jones. *Estimating Software Costs*. McGraw-Hill, Inc., New York, NY, USA, 2 edition, 2007.

[34] M. Klum. Eve online source code leaked. *http://www.neowin.net/news/eve-online-source-code-leaked*, Apr. 2008.

[35] J. Legon. Profanity, partner's name hidden in leaked microsoft code. *http://articles.cnn.com/2004-02-13/tech/microsoft.source_1_mike-gullard-windows-code-source-code?_s=PM:TECH*, Feb. 2004.

[36] R. Lemos. Cisco investigates source code leak. *http://www.techrepublic.com/article/cisco-investigates-source-code-leak/5213772*, May 2004.

[37] T. Li and N. Li. On the tradeoff between privacy and utility in data publishing. In *KDD '09*, pages 517–526, New York, NY, USA, 2009. ACM.

[38] R. F. Lorch and E. J. Oï£¡Brien, editors. *Sources of coherence in reading*. Erlbaum, Hillsdale, NJ, 1995.

[39] J. I. Maletic and A. Marcus. Supporting program comprehension using semantic and structural information. In

ICSE'01, pages 103–112, Toronto, Ontario, Canada, 2001. IEEE.

[40] A. Marcus and D. Poshyvanyk. The conceptual cohesion of classes. In *ICSM'05*, pages 133–142, Washington, DC, USA, 2005. IEEE Computer Society.

[41] A. Marcus, D. Poshyvanyk, and R. Ferenc. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Trans. Softw. Eng.*, 34:287–300, March 2008.

[42] NASA. Redaction of confidential information in electronic documents. *http://www.sti.nasa.gov/publish/redaction.pdf*, Mar. 2011.

[43] W. F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, UIUC, Champaign, IL, USA, 1992. UMI Order No. GAX93-05645.

[44] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341, 1987.

[45] D. Poshyvanyk and A. Marcus. The conceptual coupling metrics for object-oriented systems. In *22nd IEEE ICSM*, pages 469–478, Washington, DC, USA, 2006. IEEE Computer Society.

[46] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimóthy. Using information retrieval based coupling measures for impact analysis. *Empirical Softw. Engg.*, 14:5–32, February 2009.

[47] M. Revelle, M. Gethers, and D. Poshyvanyk. Using structural and textual information to capture feature coupling in object-oriented software. *Empirical Software Engineering*, 16(6):773–811, 2011.

[48] J. Richards. Facebook source code leaked onto internet. *http://www.foxnews.com/story/0,2933,293115,00.html*, June 2008.

[49] M. Schäfer, T. Ekman, and O. de Moor. Sound and extensible renaming for java. In *OOPSLA '08*, pages 277–294, New York, NY, USA, 2008. ACM.

[50] L. Seltzer. Source code leak offers novel security test. *http://www.eweek.com/c/a/Security/Source-Code-Leak-Offers-Novel-Security-Test*, Feb. 2004.

[51] I. Shield. International data privacy laws. *http://www.informationshield.com/intprivacylaws.html*, 2010.

[52] M. Sokolova, K. El Emam, S. Rose, S. Chowdhury, E. Neri, E. Jonker, and L. Peyton. Personal health information leak prevention in heterogeneous texts. In *AdaptLRTtoND '09*, pages 58–69, Stroudsburg, PA, USA, 2009. ACL.

[53] J. Staddon, P. Golle, and B. Zimny. Web-based inference detection. In *16th USENIX Security Symposium*, pages 6:1–6:16, Berkeley, CA, USA, 2007. USENIX Association.

[54] P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[55] K. Taneja, M. Grechanik, R. Ghani, and T. Xie. Testing software in age of data privacy: a balancing act. In *SIGSOFT FSE*, pages 201–211, NY, NY, USA, 2011. ACM.

[56] B. G. Thompson. *H.R.6423: Homeland Security Cyber and Physical Infrastructure Protection Act of 2010*. U.S.House, 111th Congress, Nov. 2010.

[57] F. Tip, A. Kiezun, and D. Bäumer. Refactoring for generalization using type constraints. In *OOPSLA '03*, pages 13–26, New York, NY, USA, 2003. ACM.

[58] A. Von Mayrhauser and A. Vans. Program understanding - a survey. Technical Report CS-94-120, Department of Computer Science, Colorado State University, August 23 1994. .pdf.

[59] L. Willenborg and T. d. Waal. *Elements of Statistical Disclosure Control*. Springer, NY, NY, USA, 2001.

[60] U. Yair. Five tips that can protect your company from an embarrassing leak of confidential information. *http://www.gtbtechnologies.com/Downloads/5_tips_for_protecting_data.pdf*, Mar. 2011.

[61] K. Zetter. Security breach: Tsa leaks sensitive airport screening procedure by failing to properly redact pdf. *http://www.portfolio.com/business-travel/2009/12/08/tsa-leaks-sensitive-airport-screening-manual*, Dec. 2009.

[62] K. Zetter. Goldman sachs programmer sentenced to 8 years in prison for code theft. *http://www.wired.com/threatlevel/2011/03/aleynikov-sentencing*, Mar. 2011.