Lee JB, Lee T, In HP. Topic modeling based warning prioritization from change sets of software repository. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 35(6): 1461–1479 Nov. 2020. DOI 10.1007/s11390-020-0047-8

Topic Modeling Based Warning Prioritization from Change Sets of Software Repository

Jung-Been Lee[1], Taek Lee[2], and Hoh Peter In[1,*]

[1] *College of Informatics, Korea University, Seoul 02841, Korea*
[2] *College of Knowledge-Based Services Engineering, Sungshin University, Seoul 02844, Korea*

E-mail: jungbini@korea.ac.kr; comtaek@sungshin.ac.kr; hoh_in@korea.ac.kr

Received September 19, 2019; revised May 1, 2020.

Abstract Many existing warning prioritization techniques seek to reorder the static analysis warnings such that true positives are provided first. However, excessive amount of time is required therein to investigate and fix prioritized warnings because some are not actually true positives or are irrelevant to the code context and topic. In this paper, we propose a warning prioritization technique that reflects various latent topics from bug-related code blocks. Our main aim is to build a prioritization model that comprises separate warning priorities depending on the topic of the change sets to identify the number of true positive warnings. For the performance evaluation of the proposed model, we employ a performance metric called warning detection rate, widely used in many warning prioritization studies, and compare the proposed model with other competitive techniques. Additionally, the effectiveness of our model is verified via the application of our technique to eight industrial projects of a real global company.

Keywords automated static analysis, topic modeling, warning prioritization

1 Introduction

Automatic static analysis tools find potential bugs [1] in source code during or after the software development phase. Nowadays, static analysis tools generate many warnings in software projects, but determining which warnings should be removed or fixed is difficult because the percentage of false positives ranges from 35% to 91% of the warnings reported by the tools [2, 3]. Identifying these warnings could result in developers wasting 3.6–9.5 days [1] inspecting and fixing false positives, which increase the overhead in their development.

To solve these problems, warning prioritization techniques utilize software information in addition to the automatic static analysis result. These techniques assist the automatic static analysis tools by assigning warnings in the order of defect-proneness with approaches using additional information (e.g., lines of code, revision history, and warning type). Because these techniques find the information besides that obtained through static analysis and prioritize warnings automatically, the techniques are more efficient for assigning the number of true positive warnings to the first order in its warning priority list.

However, the existing techniques [4–6] still have a problem about a level of granularity for fixing the warning category. To fix a certain warning category in the techniques, the developers should inspect all warning instances of the warning category through every source file. In the source file level inspection, despite of many false positive warning instances of the category, developers cannot stop to inspect until all warning instances are fixed in each file. Furthermore, developers spend much time in fixing many false positive warnings not related to the characteristic of the file. For example, warnings related to code complexity, such as cyclomatic complexity, may not be significant in GUI-related files because developers aptly implement anonymous classes,

Regular Paper

The research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning under Grant No. NRF-2019R1A2C2084158, and Samsung Electronics Co. Ltd.

*Corresponding Author

making codes such as event listeners more concise; however, they also result in increased code complexity, making the code difficult to read. In other words, the warnings related to code complexity are false positive warnings in GUI-related files. Because of budget limitations and schedule restrictions, the existing techniques based on the coarse-grained (i.e., file-level) inspection without the characteristic of the files still become a burden to the developers who have time to fix only a few warnings.

To overcome the abovementioned problems, we propose a topic modeling based warning prioritization technique (TOP) from change sets of a software repository. Change sets represent commits from the software repository, consisting of the changed lines (e.g., removed and added) for several purposes such as bug fixing, code refactoring at a specific date and time [7, 8]. Specially, our technique prioritizes warnings from the change sets called "bug-related code (BRC) blocks" rather than files and reflects various topics of the code blocks as one of its prioritization criteria. To identify the code blocks' topic, we use topic models, which are used to analyze large volumes of program elements and revision history in many software engineering tasks [9–14]. In particular, we use Latent Dirichlet Allocation (LDA) [15] among these topic models because LDA helps identify topics of all source code in a software project and group the source code into several topics. Using this process, our TOP technique provides an individual warning prioritization model for each topic by measuring fixed warning instances in code change blocks and topic distribution across the code blocks. If a warning is fixed frequently in a particular topic, the weight of the warning is increased in the prioritization model for the said topic. Thus, in the warning inspection stage, developers can prioritize which warnings should be fixed first according to the topic of the code blocks changed by them. Consequently, our technique provides more fine-grained warning inspection (i.e., code block level) with consideration for the characteristic (i.e., topic) of the source files.

To evaluate the performance of our prioritization technique, we apply TOP to 27 open source projects and compare it with the other techniques using the warning detection rate (WDR), which is one of the measurements to evaluate the performance of warning prioritization technique [3]. The WDR cumulates the percentage of true positive warnings detected against the number of inspections. Our evaluation results demonstrate that TOP significantly improves warning detection rate and the area of TOP's WDR is wider than those of other techniques. Namely, developers using TOP can fix warnings more quickly than those using other techniques, in the limited time available to them.

In addition to open-source projects, we apply TOP to eight industrial projects of the Lotte Data Communication Company (LDCC) to verify its usefulness in real projects. First, in comparisons among the warning prioritization techniques, i.e., area of WDR, TOP outperforms the other techniques in all projects. Second, we perform Monte Carlo simulations to measure bug-fixing time when applying our technique in comparison with the other techniques. The simulation results in that the developers using TOP could virtually save 32 minutes–90 minutes in comparison with those using other prioritization techniques, to fix 25% of the warning instances.

The remainder of this paper is organized as follows. Section 2 details TOP. Section 3 describes our experimental settings and includes an introduction of the subject programs, methods, and existing prioritization techniques for comparison. Section 4 presents the results of comparisons. Section 5 analyzes the effectiveness of our technique in industrial projects and presents detailed results. Section 6 discusses the time-cost of our technique and its application to small projects. Section 7 discusses related work, and Section 8 describes potential threats to the validity of our study. Finally, in Section 9, we conclude our study and outline future research directions.

2 Topic-Based Warning Prioritization (TOP)

This section describes the process used by TOP for prioritizing warnings from automated static analysis tools. Fig.1 shows the overall process of our proposed technique including data collection, topic modeling and warning identification, and warning category prioritization.

In the data collection phase, we identify changed code blocks related to bug-fixing, called bug-related code (BRC) blocks, in the revision history. In the topic modeling and warning identification phase, topic contributions to the topics of each BRC block are identified. In the same phase, the number of removed warning instances between two revisions is measured from the BRC blocks. In the warning category prioritization phase, we generate a list of prioritized warning categories by weighting the warning categories using topic contributions and the number of removed warning instances.

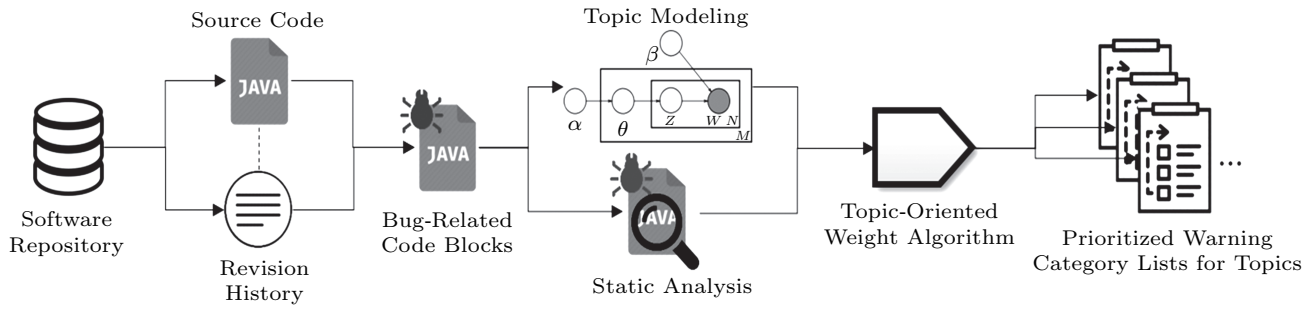

Fig.1. Overall topic-based warning prioritization (TOP) process. (a) Data collection phase. (b) Topic modeling and warning identification phase. (c) Warning category prioritization phase.

2.1 Data Collection

In this phase, we collect the source files used for training and testing datasets in our study. The inputs for this phase are the revision histories (i.e., changesets) of software repositories, such as Subversion or Git. The outputs of this phase are BRC blocks, which represent changed code blocks for fixing bugs in revisions.

There are three steps for identifying BRC blocks in this phase. First, we identify bug-related revision files by searching for two keywords, such as "Bug" and "Fix" in the commit log history [4, 16] within the revision history. Second, we identify two versions of files called Buggy and Clean, which are files from before and after fixing of bugs in each commit history, respectively. Finally, we identify changed code blocks in the Buggy and Clean files by using Git's "-diff" option. For example, in the results of the option "@@ -80, 4 + 80, 10 @@," "-80, 4" indicates that four lines have been changed starting from line 80 in the Buggy file. The changed line number in the Clean file is measured from "+80, 10" in the same manner.

Table 1 lists the commit and revision numbers of the Buggy and Clean files, as well as the BRC for the Cassandra project. For example, Cassandr.java is listed at Commit #1, and its BRC block in the Buggy file is identified from line 55 to line 65 at revision #000d4d9. Additionally, we treat files with the same name in different commits as unique files because their pieces of code are changed by different bugs in each commit. In Table 1, the Cassandra.java files collected from Commit #1 and Commit #2 are considered as two different files and renamed as [000d4d9/55-65] Cassandra.java and [012159e/43-78] Cassandra.java, respectively, based on the changed code blocks.

2.2 Topic Modeling and Warning Identification

In this phase, we perform topic modeling and static analysis, using the BRC blocks collected in Subsection 2.1 as input data.

2.2.1 *Topic Modeling*

The inputs for the topic modeling step are BRC blocks in Buggy files from the outputs of Subsection 2.1 because these blocks include bugs existing prior to revision. In this step, LDA produces topic contribution to BRC blocks to identify the topics of the blocks. For this step, we use a topic modeling tool called MALLET①. To improve the quality of topic modeling results, some

Table 1. Information for Identifying BRC Blocks for the Cassandra Project

Commit Number	File Name	Buggy File		Clean File	
		Revision Number	Changed Code Block	Revision Number	Changed Code Block
1	Cassandra.java	000d4d9	55–65	f70bdcf	55–102
2	Cassandra.java	012159e	43–78	84e6032	43–60
3	Util.java	006d5f8	10–15	518c045	10–20
			25–42		25–43
4	UUIDGen.java	0089ba8	11–102	1a1e902	11–50

①http://mallet.cs.umass.edu, Sept. 2020.

meaningless words such as stop words provided by tools, Java keywords, and package names, symbols (0–9), punctuation (.,), and special letters (!@#$%^&*) are all removed from the BRC blocks. Because Java keywords and package names are common words in all source files, these keywords do not become discrimination words. Therefore, we remove stop words in the blocks using MALLET's "–remove-stopwords" and "–extra-stopwords" options during preprocessing.

After the preprocessing, the BRC blocks are imported into MALLET's internal corpus and used to generate topic models. Before we build the topic model, the number of topics K is set appropriately in order to identify distinguishable topics. Table 2 and Table 3 present an example of the output for this step, which contains topic contributions for each BRC block and associated words for topics generated by MALLET for the Cassandra project respectively.

In the topic contribution results shown in Table 2, topic IDs and contributions are enumerated in order of contribution size within each row by the number K for all BRC blocks. Specifically, the topic ID in the second column represents the most dominant topic in each BRC block, and the third column indicates the extent to which the topic in the second column contributes to the BRC block. Table 3 presents the associated words for each topic as the result of topic modeling. These are collections of words that frequently occur together in BRC blocks associated with the corresponding topic. Additionally, the word on the left side of the second column has higher percentages of occurrence than those on the right side. For example, for [012159e/43-78] Cassandra.java in Table 2, the most prominent and the second most prominent topics and their contributions to the BRC block's content are 2 (97.05%) and 4 (0.92%), respectively. This means that the BRC block is more closely related to topic 2 than the other topics because the associated words for topic 2, such as data, column, size, key, and index, frequently occur together in the corresponding block. These topic contributions are used as weight adjustment factors for our prioritization model.

2.2.2 Static Analysis

In this step, we use PMD and identify the numbers of total and removed warning instances for the BRC blocks in Buggy and Clean files, respectively, which are collected from Subsection 2.1. PMD, a widely used automatic static analysis tool to detect potential bugs such as anti-pattern and coding standard violations, is used to report warning violations in source code of both Buggy and Clean files. In the report of warning violations, the warning category is a type of warning supported by static analysis tools. A warning instance is a single violation of the category. In the report of Buggy files, the total number of warning instances for each warning category is counted before fixing bugs, whereas the number of removed warning instances is counted after fixing bugs in the report of Clean files. For this step, any static analysis tools (e.g., Findbugs, CodeSonar, and CheckStyles) can be used depending on the program language or types of warnings to prioritize even if the tool is not PMD.

Table 4 lists the numbers of removed and total warning instances identified in the BRC blocks of Clean and Buggy files for each warning category in the Cassandra project. For example, for [000d4d9/55-65] Cassandra.java, five and 12 warning instances are removed in the Clean file based on 20 and 13 warning instances in the Buggy file for warning categories 1 and 3, respectively.

2.3 Warning Category Prioritization

In this phase, we construct a warning category prioritization model using the outputs of Subsection 2.2, including topic contributions and the number of warning instances. Our warning prioritization model is based on the weight model concept [4] called the removal likelihood (RL) and the model is defined by (1), where the

Table 2. Example of Topic Contribution Results of Each BRC Block Generated by Topic Modeling

BRC Block	1st Topic ID	Contribution to 1st Topic (%)	2nd Topic ID	Contribution to 2nd Topic (%)	···
[000d4d9/55-65] Cassandra.java	0	59.56	4	39.85	
[012159e/43-78] Cassandra.java	2	97.05	4	00.92	·
[006d5f8/10-15] Util.java	0	44.75	4	41.53	·
[006d5f8/25-42] Util.java	3	88.12	2	10.89	·
[0089ba8/11-102] UUIDGen.java	1	41.04	4	30.37	
⋮	⋮	⋮	⋮	⋮	

numbers of removed warning instances and total warning instances are identified in Subsection 2.2.2.

$$RL_C(f) = \frac{\text{number of removed warning instances of } c}{\text{Total number of warning instances of } c} \text{ in } f, \quad (1)$$

Table 3. Example of Associated Words for Each Topic Generated by Topic Modeling

Topic ID	Associated Words
0	Token, end, map, key, strategy, ···
1	Message, response, logger, debug, service, ···
2	Data, column, size, key, index, ···
3	Table, column, logger, family, store, ···
4	Column, key, family, type, bytes, ···

$RL_c(f)$ indicates the removal likelihood of the warning category c in source file f. If warning instances from the category are removed many times in the source files, the weight of the warning category is increased. In contrast, if the warning instances are seldom removed, the weight of the warning category is decreased.

While the removal likelihood model weighs the warning categories by counting all removed warning instances based on a coarse-grained level (i.e., file level) regardless of the topic of the files, we count the warning instances based on a fine-grained level (i.e., BRC blocks). Furthermore, the weight of the warning category is separated into several results according to the number of latent topics of the BRC blocks. Thus, in our approach, the weight of warning category c for topic k ($= W_c[k]$) was calculated from the whole BRC block as the following equation:

$$W_c[k] = \sum_{b=1}^{n} \left(TC_k(b) \times RL_c(b)\right). \quad (2)$$

In (2), $TC_k(b)$ presents the topic k's contribution to BRC block b. Also, $RL_c(b)$ presents the removal likelihood of warning category c in b. Since the weight value becomes larger as more BRC blocks are considered, the priority of the warning category may be biased because of the number of BRC blocks. Therefore, we normalize TC through all BRC blocks for each topic k.

To facilitate better understanding of our model, Table 5 presents an example of the prioritization of three warning categories in four BRC blocks for three topics as follows. The results of Table 5 are identified from the outputs of Subsections 2.2.1 and 2.2.2, respectively. The 2nd, 3rd, and 4th columns of Table 5 shows the topic contributions (TC) and their normalized values for the BRC blocks obtained via topic modeling. The 5th, 6th, and 7th columns of Table 5 indicate removal likelihoods (RL) of three warning categories. Specifically, the number of removed warning instances during fixing bugs and the total number of warning instances before fixing bugs are counted by performing static analysis. The warning instances are separated by "/" symbol in the 5th, 6th, and 7th columns of Table 5.

Using (2), the weights of all warning categories for each topic are calculated as follows.

- Weight values of warning categories in topic 1:

$$W_1[1] = \left(0.33 \times \frac{5}{20}\right) + \left(0.17 \times \frac{3}{4}\right) + \left(0.33 \times \frac{2}{10}\right) \approx 0.28,$$

Table 4. Removed/Total Numbers of Warning Instances in BRC Blocks of Clean and Buggy Files

BRC Block	Number of Removed Warning Instances/Total Number of Warning Instances in BRC Blocks		
	Warning Category 1	Warning Category 2	Warning Category 3
[000d4d9/55-65] Cassandra.java	5/20		12/13
[012159e/43-78] Cassandra.java		4/7	
[006d5f8/10-15] Util.java	3/4		
[006d5f8/24-42] Util.java	2/10	10/20	
[0089ba8/11-102] UUIDGen.java	15/20	1/5	10/10

Table 5. Example of the Result of Topic Modeling and Static Analysis

BRC Block	Topic-Modeling Result			Static Analysis Result		
	TC_1 (norm.)	TC_2 (norm.)	TC_3 (norm.)	Warning Category 1	Warning Category 2	Warning Category 3
[000d4d9/55-65] Cassandra.java	0.50 (0.33)	0.25 (0.18)	0.25 (0.23)	5/20		12/13
[012159e/43-78] Cassandra.java	0.25 (0.17)	0.50 (0.36)	0.25 (0.23)		4/7	
[006d5f8/10-15] Util.java	0.25 (0.17)	0.25 (0.18)	0.50 (0.45)	3/4		
[006d5f8/25-42] Util.java	0.50 (0.33)	0.40 (0.28)	0.10 (0.09)	2/10	10/20	

$$W_2[1] = \left(0.17 \times \frac{4}{7}\right) + \left(0.33 \times \frac{10}{20}\right) \approx 0.26,$$
$$W_3[1] = \left(0.33 \times \frac{12}{13}\right) \approx 0.30.$$

- Weight values of warning categories in topic 2:

$$W_1[2] = \left(0.18 \times \frac{5}{20}\right) + \left(0.18 \times \frac{3}{4}\right) + \left(0.28 \times \frac{2}{10}\right) \approx 0.24,$$
$$W_2[2] = \left(0.36 \times \frac{4}{7}\right) + \left(0.28 \times \frac{10}{20}\right) \approx 0.35,$$
$$W_3[2] = \left(0.18 \times \frac{12}{13}\right) \approx 0.17.$$

- Weight values of warning categories in topic 3:

$$W_1[3] = \left(0.23 \times \frac{5}{20}\right) + \left(0.45 \times \frac{3}{4}\right) + \left(0.09 \times \frac{2}{10}\right) \approx 0.41,$$
$$W_2[3] = \left(0.23 \times \frac{4}{7}\right) + \left(0.09 \times \frac{10}{20}\right) \approx 0.18,$$
$$W_3[3] = \left(0.23 \times \frac{12}{13}\right) \approx 0.21.$$

One can see that the priorities sorted by the weight of each warning category vary within each topic, even if the warning category is the same. The prioritized warning categories sorted by weight are listed in Table 6 as the final outputs of our technique. For example, warning category 2 has the highest priority with a weight of 0.35 in topic 2, while the same warning category has the lowest priority in topic 3 with a weight of 0.18.

Table 6. Prioritized Warning Categories and the Weight Values for Each Topic

Rank	Topic 1		Topic 2		Topic 3	
	Warning Category	Weight	Warning Category	Weight	Warning Category	Weight
1	3	0.30	2	0.35	1	0.41
2	1	0.28	1	0.24	3	0.21
3	2	0.26	3	0.17	2	0.18

Consequentially, our prioritization model generates multiple weight values of each warning category according to the number of topics. If we set the number of topic K to 3 in the topic modeling step (see Subsection 2.2.1), the weight values of each warning category must be 3. The higher the weight value of the warning category in topic k, the higher the priority; this is because the developers remove the warning category in topic k more frequently than in other topics.

3 Experimental Design

This section describes our experimental setup, including subject programs, a comparison between our technique and other techniques, and descriptions of evaluation methods and metrics for comparison.

3.1 Experimental Setting

In our experiment, 27 subject programs are used to evaluate our prioritization technique. The subject programs of our study meet the following criteria: 1) open source, 2) of various domains, 3) written in Java programming language like other warning prioritization research [1]. We find bug-related commit logs and collected almost all the Java source files related to the bug in each subject program. These fixed source files are not always considered to be bug-related files. For example, some developers classify a warning that disappears during a bug fix as a true positive warning. Therefore, we omitted those source files for the following reasons:

- newly created or deleted files for next revision;
- files in which warnings were not removed or rather increased during fix changes.

Table 7 shows our subject programs collected from various open source projects (such as GitHub, Apache Software Foundation, SourceForge, Eclipse project, and individual websites). The number of revisions is the total number of bug-related revisions found by searching for keywords such as “Bug” and “Fix” from commit log messages in revision history during the revision period. Among the revision files, the bug-related files were extracted and sorted according to above rationale. We divided all BRC blocks collected from the data collection phase (Subsection 2.1) into two halves (in time order) and used the first half of BRC blocks as a training dataset for our experiment. In other words, the set of BRC blocks in the first half period (from the first revision to revision $n/2-1$) was used to train our model, and the remainder of the recent BRC blocks (from revision $n/2$ to the latest revision) was used to test prioritization models as a ground truth [4]. The time split is based on the time required to train and test our model [1, 17]. We counted the removed warning instances contained in the latter half of the BRC blocks for the bug fixes, and the warning categories were prioritized by the number of warnings removed.

For determining the number of topics K, there exist some metrics [15, 18] based on the log-likelihood function for held-out data. However, the metrics suggest setting

Table 7. Subject Programs Used in Our Experiment

Program	Software Type	Revision Period (MM/DD/YY)	Number of Revisions	Number of Bug-Related Files	Number of Bug-Related Lines($\times 10^3$)
Ant	Java build tool	01/26/00–04/10/14	4 999	2 601	393
Bonita	BPM and workflow suit	12/21/12–06/13/16	2 196	2 022	316
Cassandra	Distributed database management system	03/18/09–01/20/16	3 202	3 072	936
Checkstyle	Static code analysis tool	08/17/01–04/16/16	3 677	3 468	163
Drools	Business rules management system	01/11/06–08/18/10	5 869	3 654	941
Elasticsearch	Distributed full-text search engine	02/14/10–04/15/16	11 355	10 498	3 638
Flink	Streaming dataflow engine	12/17/10–04/15/16	5 507	5 336	599
Guava	Google core libraries for Java	01/11/10–04/15/16	2 621	2 564	487
Guice	Lightweight dependency injection framework	01/29/07–12/01/15	1 044	978	116
Hadoop	Big data processing framework	02/27/09–01/29/16	6 380	5 977	1 683
iTextPDF	iText core Java library	12/06/00–05/26/16	1 753	1 364	430
Jackrabbit	Content repository for Java platform	09/21/04–02/26/16	3 186	3 094	445
jBPM	Flexible business process management suite	10/17/10–05/27/16	2 055	1 880	278
jclouds	Multi-cloud toolkit for Java platform	01/25/11–01/29/16	3 168	3 057	320
Jenkins	Continuous integration tool	11/05/06–03/09/16	2 857	2 718	680
libGDX	Game-development application framework	03/08/10–04/02/16	8 424	8 351	1 137
Lucene	Java-based search engine	03/18/10–10/22/15	11 269	5 982	993
Mahout	Scalable machine learning algorithms	03/14/08–03/27/16	2 642	2 594	307
Maven	Build automation tool	01/27/04–01/25/16	2 279	2 246	329
Mylyn	Subsystem of Eclipse for task management	07/08/05–05/26/16	2 967	2 736	284
Neo4j	Graph database management system	07/03/07–04/06/16	6 675	6 385	9 329
OpenMap	JavaBeans-based programmer's toolkit	01/27/03–04/01/16	2 166	2 156	345
OrientDB	Distributed graph database	04/04/10–04/18/16	10 041	9 537	2 632
Pivot	Rich web applications building platform	06/19/08–03/14/16	2 469	2 307	509
Titan	Distributed graph database	04/22/12–03/04/16	2 233	2 192	474
Tomcat	Web server	06/17/05–04/18/16	9 133	8 849	2 486
WildFly	Application server	07/21/10–04/13/16	5 519	5 201	566

too many topics. The result of the measures was negatively correlated with the measures of topic quality [19]. In addition, we are unable to acquire enough source code to build a prioritization model for certain topics because a few source code files may be included in the topics. Therefore, we experimentally determined K by trying to find the minimum number of topics, K, starting from 2 to 10 until each topic had more than 10 source code files, because the increasing number of topics worsens the interpretation of the topics and does not increase the level of agreement between human subjects and the model [19] accordingly. As a result, we determined that the number of topics, K, is consistently 5 for all the open source projects. From the topic models, we built K prioritization models (see Subsection 2.3).

3.2 Method of Performance Comparison

In this subsection, we introduce a method for classifying true and false positive warnings as ground truth labels, as well as warning prioritization techniques and a metric for comparison.

3.2.1 *Ground Truth*

In this study, we adopt a widely-used method [3, 5, 20] to classify true and false positive warning categories in a test dataset of 27 open-source projects. According to the methods, a warning instance is regarded as a true positive when the warning instance is removed in a next revision. On the other hand, if the warning instance is still remained in the next revision, the warning instance is regarded as a false positive. Although each warning instance was labeled as true or false, we use the accumulated number of warning instances in the following process:

- selection of revisions across a subject program's history starting from the first revision in the test dataset defined in Subsection 3.1;
- running of an automated static analysis tool

(PMD) on each revision to generate a report of warning violations;

• counting of warning instances that are removed in later revisions: accumulation of the number of removed warning instances as actionable alerts if a warning instance is removed; accumulation of the number of remaining warning instances as unactionable alerts if a warning instance is still present in a later revision;

• summarizing of the number of removed or remaining warning instances for each warning category over all revisions: the total number of inspected warning instances is calculated by adding the removed and remaining warning instances.

Finally, we obtain the following three variables for each warning category, which are used for calculating the performance metrics described in Subsection 3.2.3:

• number of removed warning instances,

• number of remaining warning instances,

• total number of inspected warning instances.

As part of our technique, the variables are measured from the changed code blocks belonging to topics, while the variables as part of the other techniques are measured from source files. Therefore, the values of the variables in our technique differ among different topics. The topic of the blocks is identified by the MALLET inference tool②.

3.2.2 *Prioritization Techniques for Comparison*

To investigate the effectiveness of our technique, we compare the existing prioritization techniques such as HWP [4], ACP [5], ALT [6], and other prioritization methods, such as TOOL and RAND. The reason why we choose those techniques [4, 6] is that they are nominated as representative studies from the literature reviews [1, 21] about the warning prioritization technique and the studies have similar conditions with our experimental settings, which makes comparison fair and replicable.

To generate the warning category list prioritized by the HWP technique, we identify warning removal information using the method proposed in [4]. We increase the weight of a warning category when warning instances from the category are removed many times in all source files in the training datasets for each project. In contrast, the weight of a warning category is reduced if the corresponding warning instances are seldom removed.

Additionally, we calculate the lifetime of a warning instance, as discussed in [6], to generate the warning category list prioritized by the ALT technique. We count the days from the appearance of the first warning until the removal of the final instance in all revisions of each project. The counted days for the warning instances are accumulated in each warning category.

The ACP technique is a warning classification model for determining if warning instances should be resolved and classified as actionable alerts (AAs) or left alone and classified as unactionable alerts (UAs). To classify these instances as AAs and UAs, we adopt the method described in [5] to extract the patterns of the alert characteristics (ACPs), consisting of Call, New, Binary Operation, Field Access, and Catch statements in the five revisions of the train dataset over six-month intervals for each project. We construct a warning classification model using these patterns as feature vectors in a classification algorithm. To train the model, we use a machine learning utility called Weka employing a random forest classification algorithm, which guarantees reasonable performance for the alert characteristics in an evaluation study of all publicly available features for identifying actionable warnings [20].

In random (RAND) prioritization, warning categories are prioritized by randomly selecting the warning instances in our test dataset. In the case of the tool (TOOL), warning categories are prioritized by the predefined priority of the tool. If there were warning categories with the same priority, they are sorted randomly within the same priority range.

The biggest difference between TOP and the HWP, ALT, TOOL, and RAND techniques is the number of warning category lists prioritized in the outputs of the techniques. While TOP produces multiple warning category lists based on the topics in BRC blocks, as shown in Table 6, the other techniques produce only a single warning category list. Consequently, the process of TOP for inspecting a given warning category in the source files of a test dataset consists of the following steps:

• division of each file into BRC blocks;

• identification of the topics of each block using the MALLET inference tool based on the trained topic model discussed in Subsection 2.2.1;

• application of different warning priorities (as shown in Table 6) to the warning categories in each block according to the weights of the topics.

In the other techniques, there are no processes for dividing files into code blocks and identifying the topics in blocks. These techniques consistently apply the

②http://mallet.cs.umass.edu/topics.php, Oct. 2020.

weights from a single warning category list to all source files.

3.2.3 Performance Metrics for Comparison

To compare our technique with other techniques, we use the warning detection rate (WDR) metric [3] as a performance evaluation metric. This metric has been widely used in many warning prioritization studies [1, 3–5]. In our study, we use WDR to cumulate the percentage of removed warning instances out of the total number of inspected warning instances counted based on the ground truth labels discussed in Subsection 3.2.1, according to the priorities of the warning categories. The percentage of fixed warning instances increases when true positive warnings are fixed by each technique. Typically, in an actual software project, developers focus on only top 10 to top 50 warnings, in terms of priority, as they have not enough time and budget in their project. Thus, the higher the warning detection rate in the early inspection stage, the better the technique because it allows developers to find true positive warnings quickly. The area under WDR of a good technique is also wider than those of other techniques. In our experimental result, we compare the WDR curve and the area under the curve among all techniques.

To measure and compare the WDRs of prioritization techniques, we sort the warning categories by the weights in the warning category lists generated by each technique. Next, we assume that developers fix warning categories in order of weight and consider the changes in the number of fixed warning instances and the number of inspected warning instances in the test dataset as the ground truth. Based on ground truth observations, we measure WDR by accumulating the number of removed warning instances and the total number of inspected warning instances. Some results are presented in Subsection 4.1 as representative examples. Unlike the other techniques for prioritizing warning categories, ACP classifies warning instances into binary classes (i.e., AA or UA) instead of prioritizing warning categories. Therefore, we sort the warning instances for this technique based on the distribution probability of the AA class using the *distributionForInstance* class in the WEKA tool. Additionally, we randomize the order of warning instances with the same distribution probability and calculated WDR values across 100 runs to avoid any bias resulting from randomization.

4 Results

This section presents the prioritization results generated by several techniques to explain the calculation of WDR and provide intuitive performance comparisons.

4.1 Example Prioritization Results

We precede the evaluation with an example of the warning prioritization results sorted by TOP, HWP, and TOOL for the *Ant* program, as shown in Tables 8–10. The reason for HWP and TOOL being selected as comparison targets is that HWP has been determined to be the best competitor for TOP as a warning prior-

Table 8. Warning Category List Generated by TOP and the Metrics for WDR

Topic	Category Name	Weight	Number of Removed Warning Instances	Accumulated Number of Removed Warning Instances (Percentage)	Total Number of Inspected Warning Instances	Accumulated Number of Inspected Warning Instances (Percentage)
4	ProperCloneImplementation	0.544 5	0	0 (0.0)	0	0 (0.0)
1	ProperCloneImplementation	0.186 6	0	0 (0.0)	0	0 (0.0)
0	UseLocaleWithCaseConversions	0.077 0	11	11 (2.5)	14	14 (0.8)
2	UseLocaleWithCaseConversions	0.072 3	5	16 (3.7)	11	25 (1.5)
1	AvoidDuplicateLiterals	0.070 5	1	17 (3.9)	3	28 (1.6)
1	UseLocaleWithCaseConversions	0.069 7	2	19 (4.4)	2	30 (1.8)
0	AssignmentInOperand	0.063 4	39	58 (13.3)	42	72 (4.2)
4	AvoidDuplicateLiterals	0.044 6	1	59 (13.5)	4	76 (4.4)
0	EmptyCatchBlock	0.042 5	6	65 (14.9)	12	88 (5.1)
4	NullAssignment	0.038 7	2	67 (15.4)	18	106 (6.2)
0	AvoidLiteralsInIfCondition	0.036 9	11	78 (17.9)	34	140 (8.2)
0	NullAssignment	0.036 8	5	83 (19.0)	24	164 (9.6)
1	DataflowAnomalyAnalysis	0.034 1	11	94 (21.6)	174	338 (19.7)
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 9. Warning Category List Generated by HWP and the Metrics for WDR

Category Name	Weight	Number of Removed Warning Instances	Accumulated Number of Removed Warning Instances (Percentage)	Total Number of Inspected Warning Instances	Accumulated Number of Inspected Warning Instances (Percentage)
UseEqualsToCompareStrings	0.300 0	0	0 (0.0)	1	1 (0.00)
FinalizeDoesNotCallSuperFinalize	0.180 0	0	0 (0.0)	0	1 (0.00)
FinalizeShouldBeProtected	0.180 0	0	0 (0.0)	0	1 (0.00)
ProperCloneImplementation	0.136 4	0	0 (0.0)	0	1 (0.00)
AvoidBranchingStatementAsLastInLoop	0.100 0	0	0 (0.0)	1	2 (0.01)
AvoidUsingOctalValues	0.081 8	0	0 (0.0)	0	2 (0.01)
EmptyWhileStmt	0.056 3	0	0 (0.0)	0	2 (0.01)
UseLocaleWithCaseConversions	0.056 0	26	26 (6.0)	35	37 (2.20)
ImportFromSamePackage	0.052 9	0	26 (6.0)	0	37 (2.20)
MissingBreakInSwitch	0.050 0	1	27 (6.2)	2	39 (2.30)
AvoidDuplicateLiterals	0.032 6	5	32 (7.3)	15	54 (3.20)
EmptyStatementNotInLoop	0.025 0	1	33 (7.6)	1	55 (3.20)
AvoidCatchingNPE	0.024 3	0	33 (7.6)	0	55 (3.20)
NullAssignment	0.023 5	8	41 (9.4)	61	116 (6.80)
DataflowAnomalyAnalysis	0.023 4	209	250 (57.3)	816	932 (54.30)
⋮	⋮	⋮	⋮	⋮	⋮

Table 10. Warning Categories List Provided by TOOL and the Metrics for WDR

Category Name	Weight	Number of Removed Warning Instances	Accumulated Number of Removed Warning Instances (Percentage)	Total Number of Inspected Warning Instances	Accumulated Number of Inspected Warning Instances (Percentage)
DataflowAnomalyAnalysis	5	209	209 (47.9)	816	816 (47.6)
InvalidSlf4jMessageFormat	5	0	209 (47.9)	0	816 (47.6)
DoNotThrowExceptionInFinally	4	2	211 (48.4)	6	822 (47.9)
DontImportSun	4	0	211 (48.4)	0	822 (47.9)
InstantiationToGetClass	4	0	211 (48.4)	0	822 (47.9)
StringBufferInstantiationWithChar	4	0	211 (48.4)	0	822 (47.9)
AssignmentInOperand	3	39	250 (57.3)	45	867 (50.5)
AssignmentToNonFinalStatic	3	0	250 (57.3)	1	868 (50.6)
⋮	⋮	⋮	⋮	⋮	⋮

itization technique, and TOOL has the most inefficient prioritization results. The results for other techniques, such as ALT and RAND, are publicly available at the GitHub Wiki page[③]. ACP is omitted because it cannot generate warning category lists.

In each table, the warning categories are sorted by the weight values assigned by each prioritization technique. The number of removed warning instances and the total number of inspected warning instance indicate the observations values based on the ground truth. The accumulated numbers of removed and inspected warning instances represent the cumulative numbers and percentages of removed and inspected warning instances, respectively.

As shown in Table 8, TOP has a "topic" (first column) for each warning category because the weight of a warning category varies between topics, as discussed in Subsection 2.3. Therefore, the first and the second categories of ProperCloneImplementation in the table have different weight values according to different topics. In the same manner, the third and the fourth warning categories of UseLocaleWithCaseConversions are distinguished by topics (i.e., topic 0 and topic 2, respectively). The number of removed warning instances and the total number of inspected warning instances for each warning category are the accumulated values from the BRC blocks belonging to each category.

To compare the techniques in these tables, we assume that developers established a goal to fix only the top 20% of the warning instances prioritized by each technique based on time and budget limitations. For TOP, it is sufficient to inspect only 338

[③]https://github.com/TOP-public/supplementary/wiki/Supplement1, Sept. 2020.

warning instances in the 13th warning category (i.e., DataflowAnomalyAnalysis of topic 1) because approximately 20% of the warning instances are removed based on the ground truth. In contrast, HWP could not guarantee whether 20% of the warning instances are removed, until it inspected 932 warning instances in the 15th warning category (i.e., DataflowAnomalyAnalysis). For TOOL, developers only need to inspect the first warning category. However, unless they inspect 816 warning instances simultaneously, they cannot know how many true positive warning instances are removed.

4.2 Performance Comparison

Fig.2 presents WDR curves ranked by the prioritization results of each prioritization technique, including those used to obtain the observations listed in Tables 8–10. We average the WDR value over all subject programs. The horizontal axis represents the percentage of inspected warning categories sorted by the techniques, and the vertical axis represents the percentage of fixed warnings cumulatively from the prioritization.

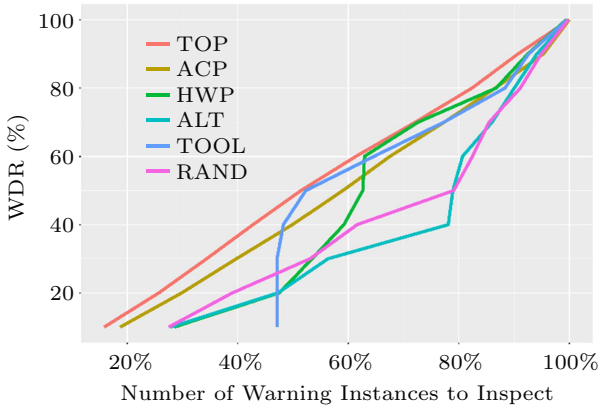

Fig.2. WDR curves for all programs until all warning instances are inspected.

As shown in Fig.2, the WDR of TOP outperforms those of the other techniques. Specifically, TOP finds nearly 40% of true positive warning instances until approximately 40% of the warning instances are inspected. In contrast, ACP finds nearly 30% of the true positive warning instances while other techniques, such as HWP, ALT, and RAND, find less than 20% of the true positive warnings. Although the WDR of ACP is greater than those of the other techniques, TOP still outperforms ACP on all warning inspections. Because many warning instances in ACP have the same distribution probabilities as the corresponding classes (i.e., AA or UA), the results of these techniques are affected by the random ordering of the warning instances. In the case of TOOL, no true positive warning instances are found in the given time, and WDR increases sharply by 50% when approximately 50% of warning instances are inspected. This is because a large number of true positive warning instances are removed previously, and approximately 50% of the total warning instances must be inspected before any additional true positives are found (see DataflowAnomalyAnalysis in Table 10).

Fig.3 shows the results of the areas under WDR curve shown in Fig.2 among all techniques for all the subject programs. The horizontal and vertical axes represent each technique and their corresponding WDR areas, respectively.

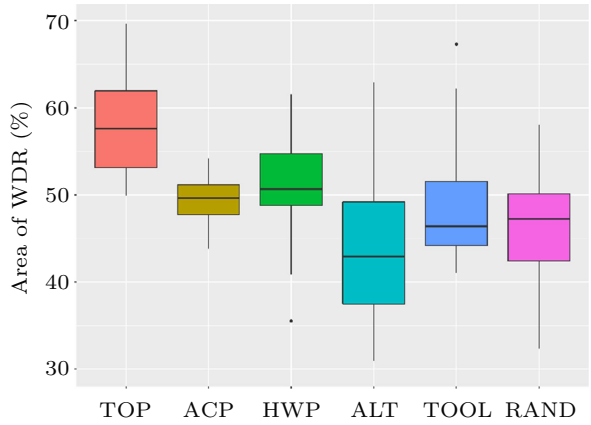

Fig.3. Area under WDR for all programs until all warning instances are inspected.

As shown in Fig.3, on average, the area of TOP is close to 60% of the WDR curve and larger than the areas of other techniques. The results for all techniques on all 27 projects are publicly available at the GitHub Wiki page[④]. The area of TOP is approximately 9%, 8%, 15%, 10%, and 13% wider than that of the ACP, HWP, ALT, TOOL, and RAND techniques, respectively. The distribution of ACP is more stable than those of other techniques because ACP guarantees similar performance for most projects. We confirm that the differences between TOP and the other techniques are statistically significant (p-value $< 0.000\,1$) based on a Wilcoxon signed rank test performed using the R statistical software. The results show that TOP is quicker at finding the true positive warnings than the other techniques with the least effort.

④https://github.com/TOP-public/supplementary/wiki/Supplement2, Sept. 2020.

5 Application to Industrial Projects

In this section, we analyze the effectiveness of our warning prioritization technique by applying the TOP technique to eight industrial projects, which are executed at Lotte Data Communication Company (LDCC)[⑤], software integration and system management parties of Korea, a leading developer of ERP projects of global company Lotte, and a leading developer of Korea government projects. Table 11 lists the industrial projects for our case study.

For the industrial projects, we find and classify revisions by searching for Korean words related to "bug" and "fix" because all commit log messages are specified in Korean. Thus, we have to double-check the log messages of all projects manually to determine if the meaning of the word in the log messages is really related to "bug" and "fix". Because we are unable to access the source code based on project constraints (i.e., schedules and policies), we could not conduct an evaluation of ACP. This process was conducted with one practitioner who has over 20 years' experience and another developer that is a CVS manager at LDCC.

Most of the revisions of the first five projects had bug-related files because the revisions were collected for the development stage. On the other hand, the revisions of the last three projects had relatively less bug-related files because these revisions were collected for the maintenance stage after the projects are released to the customer. Overall, the volume of the dataset is small, and fewer developers participated in the projects as excluding the last three projects. These projects are short-term intensive projects unlike our subject programs in Table 7. However, we think that the environment is more practical than that of an open source project for applying our technique.

5.1 Performance Comparison

Fig.4 shows the result of the area under WDR for all the subject projects presented in Table 11. Although the total number of warning instances (average of approximately 301 000 instances) is smaller than those of the open-source programs (average of approximately 2 201 000 instances) mentioned in Section 3, the WDR value is similar throughout all techniques.

The area under WDR of TOP also outperforms those of other techniques and the area of TOP is about 10%, 17%, 15%, and 7% wider than that of HWP, ALT, TOOL, and RAND, respectively, in all the projects. Although the difference between TOP and HWP is statistically insignificant (p-value $\geqslant 0.10$), the differences between TOP and the other techniques are statistically significant (p-value $\leqslant 0.01$) based on a Wilcoxon signed-rank test using the R statistical software. The result shows that developers using our TOP could have found the true positive warnings more quickly than those using the other techniques.

5.2 Simulation of Bug Fixing Time

Because a high-performance prioritization model should guarantee the detection of most warnings with a low cost, measuring the time to fix bugs is of primary importance [22–27]. To simulate bug fixing time, we measure WDR, as shown in Subsection 3.2.3. We measure the average WDR for all industrial projects based on observations of warning category lists, similar to those listed in Tables 8–10.

Table 11. Subject Projects for the Analysis of Effectiveness

Project	Software Type	Revision Period (MM/DD/YY)	Number of Bug-Revisions	Number of Bug-Related Files	Number of Bug-Related Lines ($\times 10^3$)
L.Message	Messaging integration platform	09/21/16–07/12/17	739	932	24
L.Thing-platform	Virtual data management core for IoT platform	09/23/15–02/21/17	665	659	25
L.Thing IoT Busan	Virtual data management core for IoT platform (Busan Version)	07/04/16-09/29/16	470	466	12
L.Thing-IoT Portal	Control website for IoT platform	12/01/16–05/22/17	305	304	9
LEMP	Native app. & web-based hybrid mobile platform	09/19/16–04/26/17	2 183	2 183	47
Fcss	Field consultant support system for convenience store	05/16/07-07/24/17	2 603	1 219	38
K7	Point of Sale (POS) system	02/14/06-07/21/17	3 765	439	13
K7MDC	Merchandiser (MD) support system	08/14/08-01/22/16	2 602	167	11

[⑤]https://www.idcc.co.kr/, Sept. 2020.

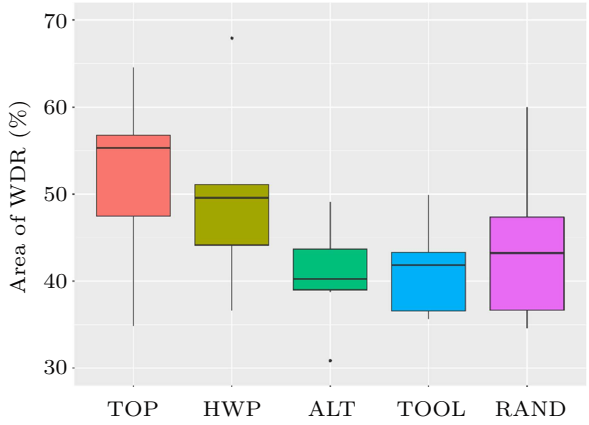

Fig.4. Area under WDR for all programs until all warning instances are inspected.

For example, as shown in Fig.5, TOP could find nearly 25% of true positive warning instances by inspecting approximately 150 warning instances while HWP, ALT, TOOL, and RAND need to inspect approximately 225–300 warning instances.

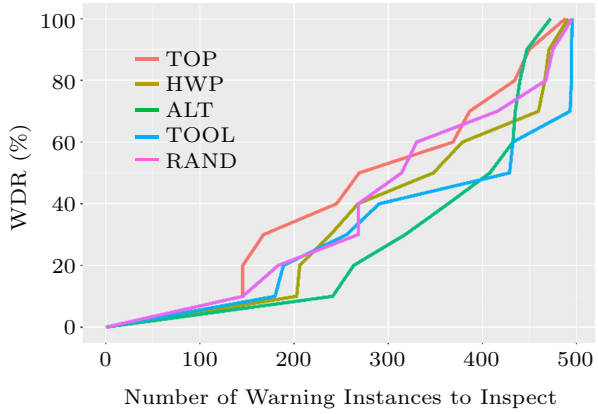

Fig.5. WDR curve for all programs until all warning instances are inspected.

We assume that if developers inspect 500 warnings, they can find 100% of the true positive warnings eventually. However, sufficient time is not available to inspect 500 warnings. As shown in Fig.5, ALT can inspect approximately 300 warnings to find 25% of the true positive warnings, assuming an inspection time of T. However, under the same conditions, TOP only needs to inspect approximately 150 warnings to find 25% of the true positives. Therefore, the inspection time for TOP can be as low as $T/2$ if we assume that time consumption increases linearly with the number of warning instances investigated.

To measure the time required to inspect warning categories, we conduct an online survey using Google Forms to determine the average decision time for inspecting a warning instance for eight representative developers from the corresponding projects. In our survey, we present the 10 most-violated warning categories for each project and their references (i.e., a summary and some examples) for inspection on the PMD website[6]. We use decision time as response options with the following values: under 10 seconds, 30 seconds, 1 minute, 2 minutes, 3 minutes, 4 minutes, 5 minutes, and over 5 minutes. Table 12 lists the results for the decision time for each developer and the project they participated in. As shown in Table 12, some developers participated in multiple projects.

Table 12. Decision Time to Fix a Warning Instance per Developer

Developer	Project	Decision Time (s)
baebon	L.Message	30
J. Y. Jang	L.Message	30
Spectacle	L.Message	120
E. Y. Oh	Fcss	30
J. S. Ryu	Fcss	30
J. W. Jung	Fcss	120
M. S. Go	L.Thing-IoT Busan	30
Kms	K7, Fcss	10

If we assume that the decision time to inspect the warning instance ranges from 10 seconds to 120 seconds as shown in Table 12, the developers using TOP may spend 25–300 minutes (150 warnings instances × 10–120 seconds) in fixing 25% of true positive warning instances. By contrast, the developers using ALT may spend as twice as much time, i.e., approximately 50–600 minutes (300 warning categories × 10–120 seconds) in fixing 25% of the true positive warning instances.

As the decision time may vary according to the developer, as shown in Table 12, we perform a Monte Carlo simulation to measure the cumulative spending time required to fix 25% the true positive warning instances for each project. Simulations conducted using Microsoft Excel 2019. Prior to conducting the simulation, we consider decision time as Kernel density estimates (KDEs) with different bandwidths within the range of 10 to 120 seconds, as shown in Table 12. Next, we generate a random value ranging from 0.0 to 1.0 in Excel to choose a decision time based on the value of the KDE for every trial. The decision time was chosen randomly, and we multiplied the decision time by the

[6]https://pmd.github.io, Sept. 2020.

number of warning instances to inspect for each technique. In case of TOP, when the decision time was set to 30 seconds, we multiplied 30 by 150 warning instances (= 4 500 seconds). We repeated this process for 100 000 trials to obtain a convergence point for the simulation results of bug fixing time. We excluded other projects from these simulations because the developers of the other projects did not answer our questions regarding decision time.

Table 13 shows the results of the Monte Carlo simulation. The results show that TOP may save more than approximately 34%, 22%, 19%, and 40% of the inspection time spent in comparison with HWP, ALT, TOOL, and RAND, respectively. It is true that other factors (e.g., usability of a given tool, and developer skillset and experience) can affect estimation results in the real world, but we focused on reporting estimations of time savings in a bug fixing scenario with the assumption that code fixing time as specified by developers would increase proportionally with the performance of the warning prioritization method.

6 Discussion

This section discusses the cost of using our technique and its application to small projects.

6.1 Cost of Using TOP

For engineers or developers who wish to apply TOP to their projects, we discuss the cost of generating TOP models in terms of computational cost and engineering effort. The steps for generating a TOP model are very simple, as discussed in Subsection 2.2.1. Regarding engineering effort, because these steps are fully automated by Python scripts codes, manual operations such as searching for log messages, parsing data, static analysis, topic modeling, and prioritizing warning categories are unnecessary. To achieve the best results, developers should determine the optimal value for the number of topics K based on iterative topic modeling, as discussed previously. However, once an optimal K is chosen for a project, no further engineering effort is required for generation of a TOP model.

We spent considerable time on data collection (Subsection 2.1), topic modeling, and static analysis (Subsection 2.2). For data collection, if source files are collected and managed well within a local SCM system, this step is much faster. For such cases, over 90% of the total time required for generating a TOP model is spent on extracting features, such as the number of warnings, and topic contributions from source files. Additional steps, such as parsing data formats and prioritizing warning categories using our weighting algorithm, account for less than 10% of the total running time. As mentioned in Subsection 2.2, for topic modeling and static analysis, the computational cost is related to the number of source files (N) collected from an SCM. Therefore, the time required for generating a TOP model increases linearly with N.

Although new source files or programs are collected and added after generating prioritized warning category lists for topics, modern static analysis tools plugged into an IDE tool can check warnings in real time. PMD used in our static analysis was also developed as an Eclipse plugin that can check source code when new code is written or added. Additionally, based on online inference algorithms [28–30], an LDA model can help update the estimates of topics when each document is observed because such models incrementally construct an updated framework. The MALLET tool used for our topic modeling process provides an option to write a serialized topic trainer object for pausing and restarting training. Therefore, developers only need to perform static analysis and topic modeling once.

6.2 Application to Small Projects

In our experiments, large open source projects were used to train prioritization models proposed based on historical data techniques, such as TOP, ACP, HWP,

Table 13. Result of the Monte-Carlo Simulation Required to Fix 25% of True Positive Warning Instances for 100 000 Trials

Program	Average of Cumulated Decision Time (min) and Standard Deviation				
	TOP	HWP	ALT	TOOL	RAND
L.Message	63 (±52)	219 (±179)	219 (±179)	79 (±66)	236 (±193)
L.Thing-IoT Busan	286 (±234)	373 (±304)	205 (±168)	373 (±305)	404 (±330)
Fcss	181 (±148)	205 (±168)	253 (±207)	205 (±168)	246 (±201)
K7MDC	5 (±4)	9 (±7)	9 (±7)	7 (±6)	8 (±6)
Average (Std.)	134 (±110)	202 (±165)	172 (±140)	166 (±136)	224 (±183)

Note: Std: standard deviation.

and ALT. Although industrial projects have less historical data than open-source projects, they still typically have sufficient data to train models. However, not all projects have enough historical data. Small and medium enterprises, and venture companies have little historical data in their SCM systems (e.g., Git or CVS) or do not even manage their code using such systems. In such circumstances, historical data based prioritization techniques will not work properly.

Nevertheless, we believe TOP still can work on projects with a small amount of historical data for any other existing TOP model that was constructed in advance with a sufficiently large historical dataset from other similar projects consisting of similar components (i.e., domains, functions, and developers). Once the topic model classifies the topics of source files in the projects, the generated warning category lists can be recycled for similar topics without any additional training processes being required. Therefore, the adaptation of an existing TOP model is useful if project similarity is verified in advance. Learning-based methods all present a similar limitation in that insufficient data results in poor performance. However, transfer learning could be a promising solution to this problem and should be investigated in future work.

6.3 Granularity Effects in LDA Application

TOP still outperforms other techniques in most subject programs using the topic model at a coarse-grained level (i.e., source file level). However, we find that WDR of TOP is insignificantly higher than some subjects. The subjects have many source files including mixed topics such as network, GUI, database in a single java class. In such a case, the quality of the topic model based on the source file is low, and MALLET inferencer cannot work well for classifying the source files into a specific topic. Thus, it is likely that wrong warning category list of TOP is applied to the files. To overcome the problem, we perform topic modeling at fine-grained level (i.e., code block level). In addition, once we divide the source code into code blocks, there is no need to inspect a large amount of warning instances including many false positives like coarse-grained level based techniques such as HWP, TOOL as described in the example of Subsection 4.1.

7 Related Work

In this section, we provide some context about the application of topic models to software engineering and explore some studies on warning prioritization.

7.1 Topic Model Application to Software Engineering

Topic models, in particular Latent Semantic Indexing (LSI) [31, 32] and LDA [15], are a type of information retrieval (IR) methods for analyzing textual information. They provide a simple way to analyze large volumes of unstructured text. A "topic" is composed of a group of words that frequently occur together. Topic models can link words with similar meanings and distinguish between the use of words with multiple meanings [33]. Among the models, LDA is a probabilistic statistical model that estimates the distributions of latent topics from textual documents. LDA uses the co-occurrence of terms in a text corpus to identify the latent structure of topics in the corpus.

Recently, topic models have been used extensively to extract topics in many software engineering tasks, as topic models require no training data, which makes them easy to use in practical settings. In addition, topic models do not require expensive data acquisition and preparation cost because they use unstructured text. Finally, topic models have proven to be fast and scalable to millions of documents or more [34].

Topic models accomplish these tasks by discovering a set of topics within the unstructured information from software repositories including the issue tracking system, communication archives, source control management (SCM) system, etc., as well as the source code [35].

There is a lot of irrelevant information in the software repositories and hence it is necessary to implement the maintenance request on the current system. For improving the maintenance task, MSR4SM [35] uses the topic model from software repositories and extracts the relevant information from each software repository based on the maintenance request and the current system.

For understanding the functional intent and overview of the software, LDA-based business domain topic extraction method [9] identifies some of the domain topics from the source code and assists the developer in comprehending large software systems. Topic_{XP} [10] also supports developers in gaining an overview of a software system by extracting and visualizing the natural language topics from the source code identifier names and comments using LDA. Similar to these studies, LDA-based feature location technique

(FLT) [13] identifies source code entities that implement a functionality. Through the result of the case study, they offer specific recommendations for configuring the LDA-based FLT.

For automatic bug localization, LDA-based static technique [11] investigates whether it is suitable for use with the open-source software systems of varying sizes and evaluates its effectiveness as compared with LSI. In the study, it is concluded that the LDA-based technique is widely applicable because its accuracy has no significant relationship with the size of the subject software system or the stability of its source code base. BugScout [12] reduces the efforts of searching through a large number of files in a project to locate the buggy code by narrowing the search space of buggy files when the developers are assigned to address bug reports.

In the study for the visualization of topic similarity from the source code, LDA-based concepts extraction approach [14] analyzes and visualizes source file similarity by computing the tangling and scattering of the extracted concepts. Also, Semantic Clustering technique based on LSI [36] visualizes how the topics found in the source code are distributed over the system by clustering to group source artifacts.

Recently, LDA has been used for bug report assignment. Entropy optimized LDA (En-LDA) [37] proposes entropy to optimize the number of topics of the LDA model and also uses it to capture the expertise and interest of developers in bug assignments. A reviewer recommendation approach based on LDA [7] generates the review expertise of developers from the topics of source code changes in the review history of a software repository. Then, the approach computes the review scores of the developers for recommending the existing reviewers.

There are no warning prioritization methods [1, 21] using topic modeling as in LSI and LDA, to the best of our knowledge. Our prioritization technique is the first one to focus on warning prioritization based on the topic model of the source code in software engineering.

7.2 Warning Prioritization and Classification

Dealing with a large number of daily bug reports from the bug tracking systems requires considerable time and resources. This leads to delay in the removal of important bugs. In software maintenance, bug triaging process analyzes these bug reports to determine whether the bugs are important or not and who will fix them [38–40].

Bug prioritization is similar to the bug triaging process. In bug prioritization, most of the studies prioritize bug reports from the bug tracking system using many techniques such as machine learning, information retrieval, clustering, mathematical, and statistical models in order to overcome the incorrect bug reports, and warning prioritization [21]. However, these studies differ from our technique in that the subject of prioritization is of high level, i.e., bug reports, in software maintenance. Our technique prioritizes the warning categories from static analysis in the source code level.

Warning prioritization is a kind of actionable alert identification technique (AAIT) classified in [1] to support ASA tools. These techniques reduce excessive fault positive warnings and increase the priority of true positive warnings in the list of reported warnings. Most of the techniques employ mathematical and statistical models and machine learning approaches.

Machine learning techniques identify patterns based on implicit and unknown—but potentially useful—information. Such patterns are used to prioritize [41, 42] and classify [5, 20] warnings as true positive or false positive warnings based on the information regarding the results of static analysis, alert characteristics, and related source code. While prioritization techniques sort warning categories based on historical data, such as removal likelihood and the time period of warning removal, classification techniques identify actionable warnings by learning the patterns of alert characteristics, such as the names and types of files, classes, methods, and the corresponding change histories. Although warning classification techniques are typically used to determine if a warning instance is actionable, such techniques perform well in our warning prioritization method by enabling sorting of warning instances based on their distribution probabilities. In particular, the statistical approaches [3, 4, 6, 43–45] build a linear model to prioritize true positive warnings. These techniques prioritize warnings by promoting or demoting from initial priority such as tool supported priority to weighted priority using a statistical model.

However, these techniques consider that all of the source code has a single topic. If the source code is separated by some topics, the attributes are also classified into suitable metrics or criteria to prioritize warnings according to the topics. The proposed TOP is the first approach to prioritizing warnings using a separate topic prioritization model classified by the source code topic.

The techniques also use additional information, called software artifact characteristics, to prioritize the

warnings. These artifact characteristics can be classified into five categories [1] based on the origin of the software artifact as follows:

• *Alert Characteristics* (AC): warning category (e.g., null pointer) reported from ASA tools;

• *Code Characteristics* (CC): code metrics like lines per file, cyclomatic complexity, etc.;

• *Source Code Repository Metrics* (SCR): attributes mined from the software repository (e.g., code churn, revision log messages, revision history);

• *Bugdatabase Metrics* (BDB): information from the bug database;

• *Dynamic Analyses Metrics* (DA): attributes associated with analyzing the source code during execution.

Most of the techniques commonly use attributes associated with CC, while some techniques use SCR. Our technique uses multiple software artifact characteristics such as AC and SCR because only this combination of the artifacts does not require any external information (i.e., developer survey or feedback, bug database) and program execution, including compilation. This information is an obstacle to prioritize the warnings automatically.

8 Threats to Validity

• *The number of topics, K, may be sensitive to the result of our prioritization model.* Determining the number of topics, K, is a major threat to validity (e.g., fishing for a specific result) [46]. If K is large, i.e., fine-grained topics, the quality of the topic model is improved but the evaluation results have poor quality [19]. Conversely, if K is small, the evaluation results are improved because the number of training and the test source files classified by the topic k become larger, which results in an improvement of the TOP model and the evaluation results. To find the optimal number of topics, K, we used perplexity [15] and the harmonic mean method [47] as a performance measurement method. However, the result of the measures is negatively correlated with the measures of topic quality. This is consistent with previous research [19]. Therefore, we experimentally determined K until we obtained sufficient source files for training our prioritization model. Nevertheless, we may not have sufficiently addressed the objective method to determine the number of topics, K. Therefore, additional future studies are required.

• *Subjects examined may not be representative.* Although we examined many open-source projects from various sources for comparison with those of previous studies, we intentionally chose subject programs and generalized biased topics that gave better (or worse) experimental results. Therefore, any subjects without various types of topics may not be significantly benefited from our prioritization technique.

• *No industrial case study was conducted in a real-world environment.* Although our technique outperformed the other techniques in eight industrial datasets, our experiment was not conducted using real-world projects and developers. Additionally, the results of the simulation of bug fixing time were not real observation results from developers. Therefore, the results of our prioritization techniques may not represent real industrial cases, and additional surveys and evaluations using real-world developers and projects are necessary.

9 Conclusions

In this paper, we proposed a topic modeling based warning prioritization technique (TOP) from change sets from a software repository. TOP provides multiple warning prioritization models by reflecting topic contribution to a static analysis result of BRC blocks. More specifically, the more the warning removals in BRC blocks related to a topic, the higher the priority of the warning categories in the topic.

To evaluate the performance of five warning prioritization techniques, namely TOP, ACP, HWP, ALT, RNAD, and TOOL, we applied each technique to 27 open source projects, and compared them with each other. Through the performance evaluation results, TOP proved that topic prioritization models based on the topics of the fine-grained code block level (i.e., BRC blocks) can find true positive warnings more quickly than the other prioritization techniques based on coarse-grained source file level. Additionally, we applied our technique to some industrial projects to verify its practicality. TOP still outperforms the other techniques on the considered actual projects, and a simulation of bug fixing time reveals that TOP can help developers save the inspection time by focusing on true positive warnings. However, based on the limitations of these simulations, it cannot be conclusively determined that TOP saves bug fixing time in real industrial projects.

Overall, we expect that a future warning prioritization technique will reflect the latent topics of source code to increase the detection rates of true positive from the result of the static analysis tool. To the best of our

knowledge, our technique is the first to focus on warning prioritization based on the topic model of fine-grained code block level. TOP is, thus, a first step in this direction. For future work, we will analyze the usability of TOP as an automated tool and investigate real bug fixing time for developers using the TOP tool in practice. Additionally, we will transfer our prioritization model (i.e., transfer learning) to small projects that have similar domains and functions to verify that our technique is still effective for such projects.

References

[1] Heckman S, Williams L. A systematic literature review of actionable alert identification techniques for automated static code analysis. *Information and Software Technology*, 2011, 53(4): 363-387.

[2] Csallner C, Smaragdakis Y, Xie T. DSD-Crasher: A hybrid analysis tool for bug finding. *ACM Transactions on Software Engineering and Methodology*, 2008, 17(2): Article No. 8.

[3] Heckman S, Williams L. On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In *Proc. the 2nd ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, October 2008, pp.41-50.

[4] Kim S, Ernst M D. Which warnings should I fix first? In *Proc. the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, September 2007, pp.45-54.

[5] Hanam Q, Tan L, Holmes R, Lam P. Finding patterns in static analysis alerts: Improving actionable alert ranking. In *Proc. the 11th ACM Working Conference on Mining Software Repositories*, May 2014, pp.152-161

[6] Kim S, Ernst M D. Prioritizing warning categories by analyzing software history. In *Proc. the 4th International Workshop on Mining Software Repositories*, May 2007, Article No. 27.

[7] Corley C S, Damevski K, Kraft N A. Changeset-based topic modeling of software repositories. *IEEE Transactions on Software Engineering*. doi:10.1109/TSE.2018.2874960.

[8] Corley C S, Kashuda K L, Kraft N A. Modeling changeset topics for feature location. In *Proc. the 31st IEEE International Conference on Software Maintenance and Evolution*, September 2015, pp.71-80.

[9] Rama G M, Sarkar S, Heafield K. Mining business topics in source code using latent Dirichlet allocation. In *Proc. the 1st Annual India Software Engineering Conference*, February 2008, pp.113-120.

[10] Savage T, Dit B, Gethers M, Poshyvanyk D. Topic$_{XP}$: Exploring topics in source code using latent Dirichlet allocation. In *Proc. the 26th IEEE International Conference on Software Maintenance*, September 2010.

[11] Lukins S K, Kraft N A, Etzkorn L H. Bug localization using latent Dirichlet allocation. *Information and Software Technology*, 2010, 52(9): 972-990.

[12] Nguyen A T, Nguyen T T, Al-Kofahi J, Nguyen H V, Nguyen T N. A topic-based approach for narrowing the search space of buggy files from a bug report. In *Proc. the 26th IEEE/ACM International. Conference on Automated Software Engineering*, November 2011, pp.263-272.

[13] Biggers L R, Bocovich C, Capshaw R, Eddy B P, Etzkorn L H, Kraft N A. Configuring latent Dirichlet allocation based feature location. *Empirical Software Engineering*, 2014, 19(3): 465-500.

[14] Linstead E, Rigor P, Bajracharya S, Lopes C, Baldi P. Mining concepts from code with probabilistic topic models. In *Proc. the 22nd IEEE/ACM International Conference on Automated Software Engineering*, November 2007, pp.461-464.

[15] Blei D M, Ng A Y, Jordan M I. Latent Dirichlet allocation. *Journal of Machine Learning Research*, 2003, 3: 993-1022.

[16] Mockus A, Votta L G. Identifying reasons for software changes using historic databases. In *Proc. the 16th International Conference on Software Maintenance*, October 2000, pp.120-130.

[17] Witten I, Frank E, Hall M, Pal C. Data Mining: Practical Machine Learning Tools and Techniques (4th edition). Morgan Kaufmann, 2016.

[18] Ponweiser M. Latent Dirichlet allocation in R [M.S. Thesis]. Vienna University of Economics and Business, 2012.

[19] Chang J, Gerrish S, Wang C, Boyd-Graber J L, Blei D M. Reading tea leaves: How humans interpret topic models. In *Proc. the 23rd Annual Conference on Neural Information Processing Systems*, December 2009, pp.288-296.

[20] Wang J, Wang S, Wang Q. Is there a "golden" feature set for static warning identification? An experimental evaluation. In *Proc. the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, October 2018, Article No. 17.

[21] Uddin J, Ghazali J, Deris M M, Naseem R, Shah S. A survey on bug prioritization. *Artificial Intelligence Review*, 2017, 47(2): 145-180.

[22] Rahman F, Posnett D, Hindle A, Barr E, Devanbu P. BugCache for inspections: Hit or miss? In *Proc. the 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering and 13th European Software Engineering Conference*, September 2011, pp.322-331.

[23] Hata H, Mizuno O, Kikuno T. Bug prediction based on fine-grained module histories. In *Proc. the 34th International Conference on Software Engineering*, June 2012, pp.200-210.

[24] Koru A G, Emam K E, Zhang D, Liu H, Mathew D. Theory of relative defect proneness. *Empirical Software Engineering*, 2008, 13(5): 473-498.

[25] Menzies T, Milton Z, Turhan B, Cukic B, Jiang Y, Bener A. Defect prediction from static code features: Current results, limitations, new approaches. *Automated Software Engineering*, 2010, 17(4): 375-407.

[26] Arisholm E, Briand L C, Johannessen E B. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*, 2010, 83(1): 2-17.

[27] Mende T, Koschke R. Effort-aware defect prediction models. In *Proc. the 14th European Conference on Software Maintenance and Reengineering*, March 2010, pp.107-116.

[28] AlSumait L, Barbará D, Domeniconi C. On-line LDA: Adaptive topic models for mining text streams with applications to topic detection and tracking. In *Proc. the 8th IEEE International Conference on Data Mining*, December 2008, pp.3-12.

[29] Canini K, Shi L, Griffiths T. Online inference of topics with latent Dirichlet allocation. In *Proc. the 12th International Conference on Artificial Intelligence and Statistics*, April 2009, pp.65-72.

[30] Hoffman M, Bach F R, Blei D M. Online learning for latent Dirichlet allocation. In *Proc. the 24th Annual Conference on Neural Information Processing Systems*, December 2010, pp.856-864.

[31] Deerwester S, Dumais S T, Furnas G W, Landauer T K, Harshman R. Indexing by latent semantic analysis. *Journal of the Association for Information Science and Technology*, 1990, 41(6): 391-407.

[32] Hofmann T. Probabilistic latent semantic indexing. In *Proc. the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, August 1999, pp.50-57.

[33] Steyvers M, Griffiths T. Probabilistic topic models. In *Handbook of Latent Semantic Analysis*, Landauer T, McNamara D, Dennis S, Kintsch W (eds.), Psychology Press, 2007, pp.424-440.

[34] Thomas S W. Mining software repositories using topic models. In *Proc. the 33rd International Conference on Software Engineering*, May 2011, pp.1138-1139.

[35] Sun X, Li B, Leung H, Li B, Li Y. MSR4SM: Using topic models to effectively mining software repositories for software maintenance tasks. *Information and Software Technology,* 2015, 66: 1-12.

[36] Kuhn A, Ducasse S, Gîrba T. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 2007, 49(3): 230-243.

[37] Zhang W, Cui Y, Yoshida T. En-LDA: An novel approach to automatic bug report assignment with entropy optimized latent Dirichlet allocation. *Entropy*, 2017, 19(5): Article No. 173.

[38] Moin A, Neumann G. Assisting bug triage in large open source projects using approximate string matching. In *Proc. the 7th International Conference on Software Engineering Advances*, November 2012.

[39] Murphy G, Cubranic D. Automatic bug triage using text categorization. In *Proc. the 16th International Conference on Software Engineering and Knowledge Engineering*, June 2004, pp.92-97.

[40] Jeong G, Kim S, Zimmermann T. Improving bug triage with bug tossing graphs. In *Proc. the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, August 2009, pp.111-120.

[41] Jung Y, Kim J, Shin J, Yi K. Taming false alarms from a domain-unaware C analyzer by a Bayesian statistical post analysis. In *Proc. the 12th International Conference on Static Analysis*, September 2005, pp.203-217.

[42] Yi K, Choi H, Kim J, Kim Y. An empirical study on classification methods for alarms from a bug-finding static C analyzer. *Information Processing Letters*, 2007, 102(2/3): 118-123.

[43] Ruthruff J, Penix J, Morgenthaler J, Elbaum S, Rothermel G. Predicting accurate and actionable static analysis warnings: An experimental approach. In *Proc. the 30th International Conference on Software Engineering*, May 2008, pp.341-350.

[44] Kremenek T, Engler D. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *Proc. the 10th International Conference on Static Analysis*, June 2003, pp.295-315.

[45] Kremenek T, Ashcraft K, Yang J, Engler D. Correlation exploitation in error ranking. In *Proc. the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering Notes*, October 2004, pp.83-93.

[46] Wohlin C, Runeson P, Höst M, Ohlsson M C, Regnell B, Wesslén A. Experimentation in Software Engineering. Springer Science & Business Media, 2012.

[47] Griffiths T L, Steyvers M. Finding scientific topics. In *Proc. National Academy of Sciences of the United States of America*, April 2004, pp.5228-5235.

Jung-Been Lee received his Ph.D. degree in the College of Informatics at Korea University, Seoul, in 2020. He is currently a Postdoctoral Researcher in Chronobiology Institute at Korea University, Seoul. His major areas of study are mining software artifacts and analysis, Blockchain engineering, and machine learning in bioinformatics.

Taek Lee is currently an assistant professor in College of Knowledge-Based Services Engineering at Sungshin University, Seoul. He received his Ph.D. degree in computer science and engineering at Korea University, Seoul, in 2016. His research interests include software analytics, software defect prediction, mining software repositories, healthcare-ICT convergence, and information security.

Hoh Peter In received his Ph.D. degree in computer science from the University of Southern California (USC), College Station. He was an assistant professor at Texas A&M University, College Station. At present, he is a professor in College of Informatics at Korea University, Seoul. He is an editor of the EMSE and TIIS journals. His primary research interests are software engineering, social media platform and services, and software security management. He earned the Most Influential Paper Award for 10 years in ICRE 2006. He has published over 100 research papers.