Which Crashes Should I Fix First?: Predicting Top Crashes at an Early Stage to Prioritize Debugging Efforts

Dongsun Kim, *Member*, *IEEE*, Xinming Wang, *Student Member*, *IEEE*, Sunghun Kim, *Member*, *IEEE*, Andreas Zeller, *Member*, *IEEE*, S.C. Cheung, *Senior Member*, *IEEE*, and Sooyong Park, *Member*, *IEEE*

Abstract—Many popular software systems automatically report failures back to the vendors, allowing developers to focus on the most pressing problems. However, it takes a certain period of time to assess which failures occur most frequently. In an empirical investigation of the Firefox and Thunderbird crash report databases, we found that only 10 to 20 crashes account for the large majority of crash reports; predicting these "top crashes" thus could dramatically increase software quality. By training a machine learner on the features of top crashes of past releases, we can effectively predict the top crashes well before a new release. This allows for quick resolution of the most important crashes, leading to improved user experience and better allocation of maintenance efforts.

Index Terms—Top crash, machine learning, crash reports, social network analysis, data mining.

1 Introduction

Many of today's software systems include *automated problem reporting:* As soon as a problem occurs, the system reports the problem details back to the vendor, who can then leverage these details to fix the problem. As an example of automated problem reporting, consider the well-known *Firefox* Internet browser. When the runtime or operating system (OS) detects an unrecoverable failure (a "crash"), the browser process is terminated. A separate "talkback" process detects this and requests the user to submit a *crash report* that summarizes this crash occurrence to Firefox developers (Fig. 1). Each crash report includes a *crash point*,[1] that is, the program location where the crash occurred. Crashes that have the same crash point are considered to be the same [58]. In addition, a crash report includes other information relevant to the crash occurrence, such as user comments, hardware and software configuration, as well as *thread stack traces*—stacks of methods that were active at the moment of the crash.

1. It is also called a *crash signature* [54].

- *D. Kim and S. Park are with the Department of Computer Science and Engineering, Sogang University, Shinsoo-dong, Mapo-gu, Seoul 121-742, South Korea. E-mail: {darkrsw, sypark}@sogang.ac.kr.*
- *X. Wang, S. Kim, and S.C. Cheung are with the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong. E-mail: {rubin, hunkim, scc}@cse.ust.hk.*
- *A. Zeller is with the Department of Computer Science, Saarland University, Campus E1 1, Saarbrücken 66123, Germany. E-mail: zeller@acm.org.*

Manuscript received 29 Nov. 2009; revised 26 Apr. 2010; accepted 2 June 2010; published online 7 Feb. 2011.
Recommended for acceptance by A. Bertolino.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2009-11-0384.
Digital Object Identifier no. 10.1109/TSE.2011.20.

The number of crash reports thus submitted can be large: Everyday, Firefox users submit thousands of crash reports; the number spikes considerably right after a new software release. The crashes summarized by these crash reports, however, are not equally frequent.[2] As we show in this paper, it can well be that only a small number of crashes account for the vast majority of crash reports. We call these crashes *top crashes*. In Thunderbird, for instance, we found that more than 56.42 percent of all crash reports could be traced back to only *20 crashes*. In other words, if we could fix the defects that cause these 20 "top crashes" beforehand, we could reduce the number of crash occurrences by 56.42 percent or more. In Firefox, the top-20 crashes even accounted for 78.26 percent of all crash reports, implying that a little effort could yield a big gain—if only we knew which defect to fix. It is therefore crucial to identify these top crashes as soon as possible such that developers can prioritize their debugging efforts and address the top crashes first.

Currently, most crash reporting systems sort crashes based on the number of their crash reports. These systems can easily be used to recognize the top crashes—but only in hindsight. To identify the top crashes, one must *wait and see* until enough crash reports have been submitted; this implies that users have to suffer many crashes before getting a fix, leading to possible data loss and frustration. In this paper, we thus investigate strategies to *predict* top crashes. Specifically, our goal is to determine whether a crash is a top crash *the first time it occurs*. Such a prediction can be applied to identify top crashes at an early stage of development, for example, in the alpha or beta-testing phases, when only a few crash reports are available. This may allow developers to focus on top crashes earlier and improve the overall quality of the software in a much more cost-effective manner.

2. The frequency of a crash is defined as the number of crash reports that contain the corresponding crash point.

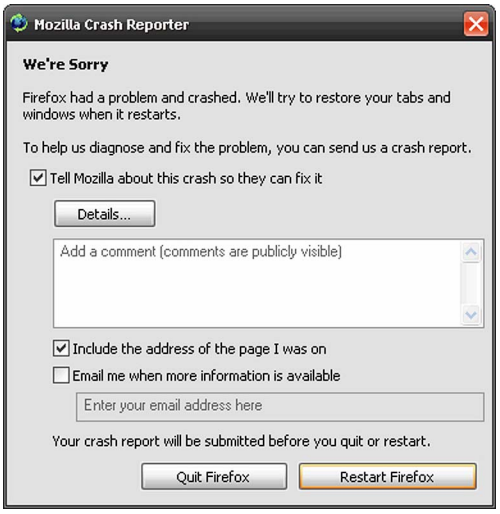


Fig. 1. A Firefox crash message from a user's perspective.

To address this challenge, we adopt a *learning-based approach*, summarized in Fig. 2. From an earlier release, we know which crash reports are "top" (frequent) and which ones are "bottom" (infrequent). We extract the top and bottom stack traces as well as their method signatures. The features of these signatures are then passed to a machine learner. The learner can then immediately classify a crash summarized by a new incoming crash report as frequent (a top crash) or not. As shown in Section 3, the deployment of an accurate top-crash predictor may reduce the number of crash reports in Firefox 3.5 by at least 36 percent if developers fix top crashes first.

We employ *features from crash reports and source code* to train a machine learner. Our preliminary observations and insights led us to focus on three types of features that form the core of our approach:

- First, we observed that statistical characteristics can indicate whether a crash is a top or bottom crash: In particular, methods in stack traces of top crashes appear again in other top crashes. This motivated us to extract *historical* features from crash reports.
- Second, intramethod characteristics can also indicate whether a method belongs to frequent crashes; complex methods may crash more often. This motivated us to employ *complexity metrics* (CM) features such as lines of code and the number of paths for top-crash prediction.
- Third, intermethod characteristics can describe crash frequency; well-connected methods in call graphs may crash often. To measure connectedness, we employ *social network analysis* (SNA) features such as centrality.

To validate our approach, we investigate the crash report repositories of the *Firefox* Web browser as well as the *Thunderbird* e-mail client. We use a very small training set of only 150-250 crash reports from a prior release (that is, the crash reports received within 10-15 minutes after release). Given the small size of the set, the machine learner can then classify crash reports for the new release *immediately*—that is, with the very first crash report. This classification method has a high accuracy: In Firefox, 75 percent of all incoming reports are correctly classified; in Thunderbird, the accuracy rises to 90 percent. These accurate prediction results can provide valuable information for developers to prioritize their defect-fixing efforts, improve quality at an early stage, and improve the overall user experience.

From a technical standpoint, this paper makes the following contributions:

1. We present a novel technique to predict whether a crash will be frequent (a "top crash") or not.
2. We evaluate our approach on the crash report repositories of Thunderbird and Mozilla, demonstrating that it scales to real-life software.
3. We show that our approach is *efficient*, as it requires only a small training set from the previous release. This implies that it can be applied at an early stage of development, e.g., during alpha or beta testing.
4. We show that our approach is *effective*, as it predicts top crashes with high accuracy. This means that effort on addressing the predicted problems is well spent.
5. We discuss and investigate under which circumstances our approach works best; in particular, we investigate which features of crash reports are most sensitive for successful prediction.

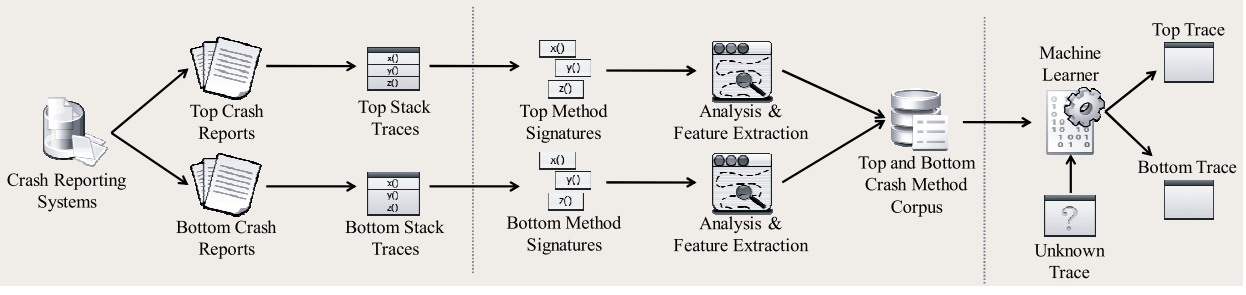


Fig. 2. Approach overview. Our approach has three steps: extracting traces from top and bottom crash reports, creating training data from the traces, and predicting unknown crashes. The first step classifies top and bottom crashes and extracts stack traces from their reports. The second step extracts methods from the stack traces and characterizes these methods using feature data, which are extracted from source code repositories. Feature values are then accumulated per trace. These are used for training a machine learner. In the prediction step, the machine learner takes an unknown crash stack trace and classifies it as a top or bottom trace. (a) Extracting crash traces. (b) Creating corpus. (c) Prediction.

6. Overall, we thus allow for quick resolution of the most pressing bugs, increasing software stability, and hence, user satisfaction.

The remainder of the paper is organized as follows: After giving details and insights into crash reporting at Firefox and Thunderbird (Sections 2 and 3), we describe our approach in detail (Section 4), investigating central items such as feature selection (FS) and model construction. Section 5 describes the evaluation of our approach on the Firefox and Thunderbird crash report repositories and presents the evaluation results. Section 6 discusses potential issues of our approach and identifies threats to its validity. After discussing the related work (Section 7), we close with conclusions and future work (Section 8).

2 Background

Many software products support crash reporting systems, such as Dr. Watson [46], Apple Crash Report [16], and Breakpad [9]. When a crash occurs, the system automatically generates a crash report that captures the program's status and then sends this crash report to the corresponding crash report repository maintained by the software development team. For example, the BreakPad [9] system used in Mozilla gathers the crash point, crash time, operating system and its version, hardware information, crash reason, crash address, optional user comments, and all thread stack traces from the crash. An example of a crash report is shown in Fig. 3.

The information provided by crash reports is valuable for bug localization and fixing. Its availability has made a huge difference in the manner in which software is developed. For instance, crash reports have enabled Microsoft to fix 29 percent of the Windows XP bugs that were covered in Windows XP SP1, and more than half of the Office XP errors bugs that were covered in Office XP SP2 [15].

Although crash reporting systems facilitate information collection, they are also likely to raise an issue: There are too many crashes for developers to investigate thoroughly [51]. For example, Firefox (all versions) users submit more than 600,000-700,000 crash reports every week. These crash reports contain 700-1,900 distinct crashes for each version. Most of them were reported within a few days after release. In the case of Firefox 3.5, which is one of the latest versions of Firefox and was released on 30 June 2009, users submitted 410,000 crash reports within one month of its release, with more than 750 distinct crashes. As shown in Fig. 4, the number of crash reports steadily increased for two weeks. More than 15,000 crash reports were submitted per day until the next version (Firefox 3.5.1) was released. Note that Firefox 3.5.1 and its descendants also show results similar to those of Firefox 3.5.

Given this large number of crashes, it is unlikely that developers would have had sufficient time to investigate all of them in a short period of time. It is in this background that we are interested in top crashes.

3 Motivation

In this section, we present the details of our investigation into Firefox and Thunderbird crashes. This investigation has shed some light on the reasons for focusing on top crashes (Section 3.1), the limitations of the current practice in identifying top crashes (Section 3.2), and how prediction of top crashes can improve the practice (Section 3.3). Note that all of the data reported in this section are gathered from the Mozilla crash repository [49], which is made publicly available in a Socorro [54] server.

Firefox 3.0.13 Crash Report [@arena_run_reg_alloc]

ID: fe4217ee-8c80-4348-a1f0-d597a2090817
Signature: arena_run_reg_alloc

Details | Modules | Raw Dump

Signature	arena_run_reg_alloc
UUID	fe4217ee-8c80-4348-a1f0-d597a2090817
Time	2009-08-17 12:09:07.584227
Uptime	0
Last Crash	156 seconds before submission
Product	Firefox
Version	3.0.13
Build ID	2009073022
Branch	1.9
OS	Windows NT
OS Version	5.1.2600 Service Pack 3
CPU	x86
CPU Info	AuthenticAMD family 15 model 79 stepping 2
Crash Reason	EXCEPTION_ACCESS_VIOLATION
Crash Address	0x1efcaf08
User Comments	put back on firefox for me
Processor Notes	

Crashing Thread

Frame	Module	Signature [Expand]
0	mozcrt19.dll	arena_run_reg_alloc
1	mozcrt19.dll	arena_malloc_small
2	mozcrt19.dll	arena_malloc
3	mozcrt19.dll	malloc
4	xul.dll	PL_DHashAllocTable
5	xul.dll	ChangeTable
6	xul.dll	PL_DHashTableOperate
7	xul.dll	nsCOMPtr_base::assign_from_qi
8	xul.dll	nsTHashtable<nsBaseHashtableET<nsISupportsHashKey,nsRefPtr<nsXM > >::PutEntry
9	xul.dll	nsBaseHashtable<nsHashableHashKey,__int64,__int64>::Put
10	xul.dll	nsComponentManagerImpl::ReadPersistentRegistry
11	xul.dll	xul.dll@0x2ddf5d
12	xul.dll	ScopedXPCOMStartup::Initialize
13	xul.dll	XRE_main
14	firefox.exe	wmain
15	firefox.exe	firefox.exe@0x217f
16	kernel32.dll	BaseProcessStart

Show/hide other threads

Fig. 3. Example of a crash report. A crash report includes the crash method signature, crashed time, OS and its version, user comments, and stack traces of all threads when the crash occurs. Developers use this information to fix crashes.

3.1 Why Focus on Top Crashes?

One of the central hypotheses in our work is that a small number of top crashes account for a majority of crash reports. To verify this hypothesis, we investigated the crash reports for Firefox 3.0 from July to December 2008 and those for Thunderbird 3.0 from January to May 2009. More than 390,000 Firefox 3.0 crash reports and 35,000 Thunderbird 3.0 crash reports were gathered. From the data, we observed that some crashes were reported more frequently than others. For example, one crash with crash point at a statement of the function "`_PR_MD_SEND`" was reported more than 11,000 times per week, while another crash with crash point at a statement of the function "`nsHTMLDocument::Release()`" was reported less than twice per week. Most crash repositories, including Socorro [54], list the most-reported crashes first. To illustrate the distribution

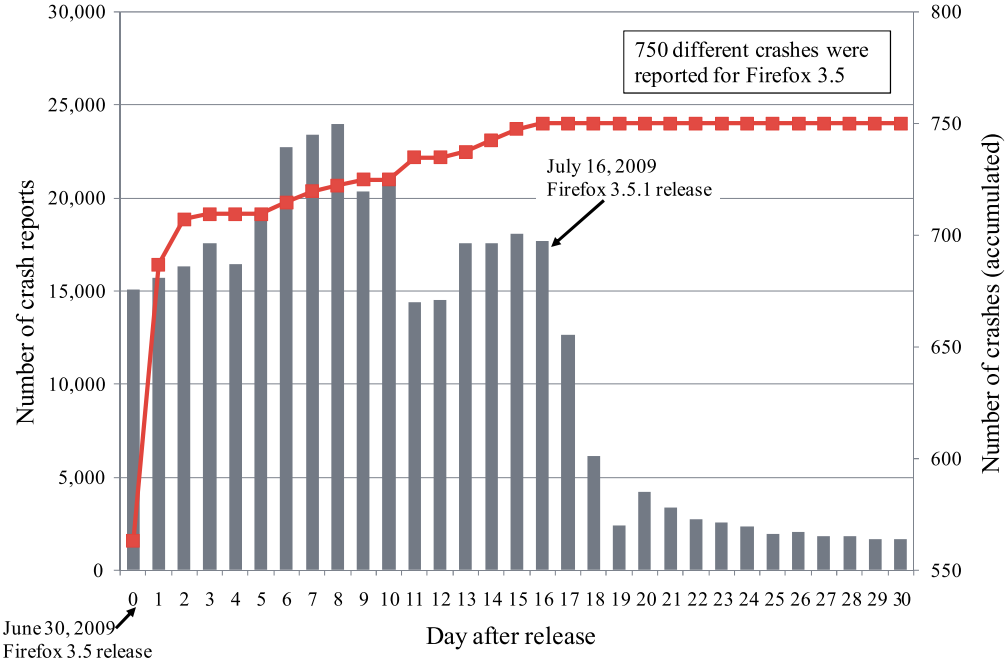


Fig. 4. Number of crash reports for Firefox 3.5 per day since its release (30 June 2009). More than 14,000-24,000 crash reports have been reported per day. The number of crash reports indicates that users experienced at least the same number of failures (abrupt program termination). Note that 750 crashes for (crash points) are reported for Firefox 3.5.

of crash reports, we sorted crashes by their frequency of being reported, and then counted the percentage of crash reports accounted for in each interval of 10 crashes. The bar chart in Fig. 5 shows the results. For example, the leftmost bar indicates that the top-10 crashes accounted for more than 50 percent of the Firefox crash reports and more than 35 percent of the Thunderbird crash reports. Fig. 5 provides the initial validation of our hypothesis: For example, the top-20 crashes account for 72 and 55 percent of the crash reports for Firefox and Thunderbird, respectively.

Note that such a trend has also been observed in commercial software. For example, by analyzing crash reporting data, Microsoft has found that a small set of defects is responsible for the vast majority of its code-related problems: "fixing 20 percent of code defects can eliminate 80 percent or more of the problems users encounter" [1]. This indicates that identifying top crashes is important for commercial products as well as open source projects.

Moreover, such a phenomenon is not restricted to crash-related failures. For example, Adams [2] observed that most operational system failures are caused by a small proportion of latent faults. Goseva and Hamill [23], [25] observed that a few small regions in a program could account for the reliability of the whole program. Our finding here is consistent with these studies.

3.2 Limitation of Current Practice

Top crashes need to be fixed as soon as possible. Given a top crash, how long does it take for developers to start working on it? Ideally, a top crash should be handled immediately once it is reported. In other words, the date of a first crash report should be close to the date when developers begin to work on the crash. To verify whether this is the case in the real world, we investigated the crashes and bug-fixing activities of Firefox 3.5.

One issue here is how to determine the time when developers begin to work with a crash. In Mozilla projects such as Firefox and Thunderbird, management policy mandates that any bug-fixing activity for a crash in the crash repository must begin with the creation of a bug report using Bugzilla [10] by the developer. Thus, when the developer creates a bug report for a crash, we assume that he or she is ready to work on this crash. Therefore, we regard the time when its corresponding bug report is created as the time when developers begin to work on this crash. With this information, we calculated the number of days it took for a

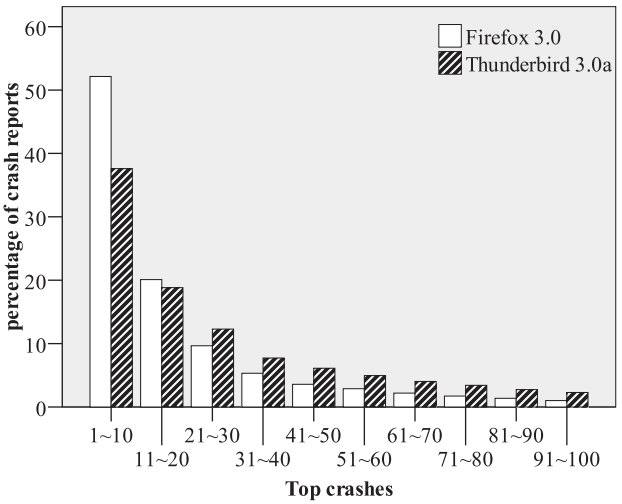


Fig. 5. Number of crash reports ranked in groups of 10 for Firefox and Thunderbird. Firefox 3.0 and Thunderbird 3.0 crash reports were collected for July 2008-December 2008, and January 2009-May 2009, respectively. The top-10 crashes accounted for more than 35 percent (Thunderbird) and 50 percent (Firefox) of the total number of crash reports.

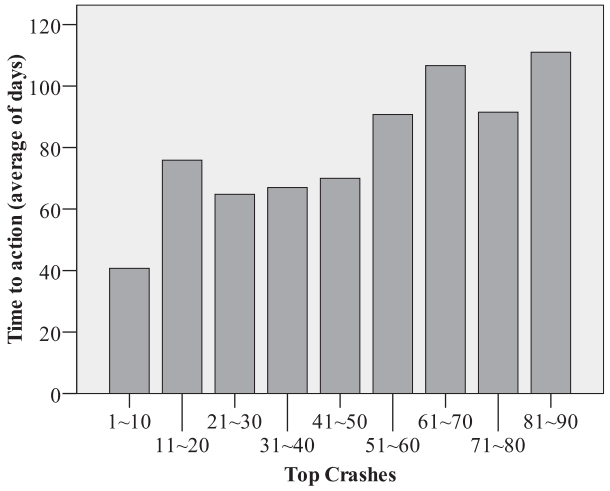


Fig. 6. Number of days for crashes to be reported as bugs (Firefox 3.5). We measured the number of days between the first crash report for each crash and its bug report. There was a correlation between the crash's ranking and time taken for bug reporting.

developer to start working on a top crash. Fig. 6 shows the results for the top-100 crashes of Firefox 3.5.

From Fig. 6, we can observe that the real situation is far from ideal: On average, developers waited 40 days until they started to work on a top-10 crash. This is unfortunate because, given the frequency of these top crashes, such a delay would mean hundreds of thousands of crash occurrences.

So why did Mozilla developers allow such a long delay in handling top crashes? One might blame this delay on insufficient motivation for maintenance. However, our personal communication with Mozilla development team members Gary Kong and Channy Yun suggests otherwise: Mozilla developers are generally eager to work on top crashes. However, they are conservative in acknowledging a crash as a top crash, even if it appears at the top of the list for the moment. This conservativeness is driven by the concern that, at the early stage when crashes are first reported (e.g., in the alpha and beta-testing phases), the frequency of a crash might be substantially different from its frequency at the later stage. Therefore, developers prefer to "wait and see" until there are sufficient crash reports to support a crash being a top crash.

What if Mozilla developers were less conservative? Let us assume that they had used the data at an early stage, the alpha-testing phase, to determine top crashes. Using the 5,199 crash reports submitted during the alpha-testing phase of Firefox 3.5, they would replace those crashes that occurred most frequently in this stage. However, are these crashes really the top crashes? Fig. 7 illustrates the ranking of these crashes in terms of their actual occurrence frequencies, which are derived from all 415,351 crash reports submitted during the main life span of Firefox 3.5 (from the start of alpha testing to the day when the next version was released). In this figure, each bar represents a k-most-frequent crash in the alpha-testing phase. For example, the leftmost bar indicates that the most-frequent crash in the alpha-testing phase is ranked 162nd in terms of actual occurrence frequency.

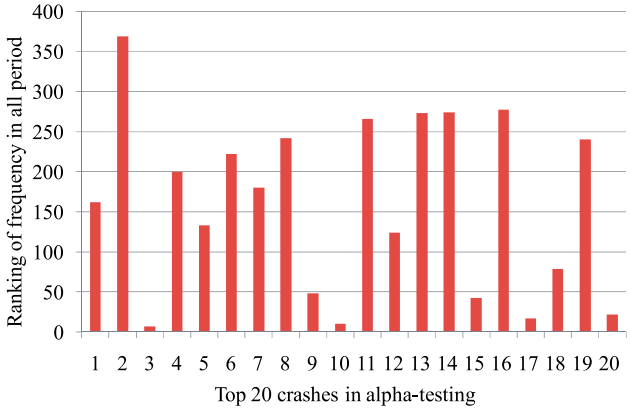


Fig. 7. The ranking of most-frequent crashes in the alpha-testing phase.

From Fig. 7, we can observe that the k-most-frequent crashes in the alpha-testing phase are poor indicators of actual top crashes: Only two of them ($k = 3$ and $k = 10$) are top-20 crashes, while most of the others are actually infrequent crashes. In fact, the 20 most-frequent crashes in the alpha-testing phase can account for only 13.35 percent of the all crash reports of Firefox 3.5, whereas the actual top-20 crashes account for 78.26 percent. The key reason, as pointed out by Fenton and Neil [19], is that the failure rate of a fault at the early stage (prerelease) can be significantly different from its failure rate after release. In practice, the goal of internal and volunteer alpha testers is to expose the most number of bugs with the least number of test cases. Therefore, they usually tend not to repeat already-exercised crashing test cases even though these test cases might trigger top crashes.

The above discussion highlights the dilemma of the current practice: By being more conservative in determining top crashes, developers delay bug fixing, but by being less conservative in determining top crashes, developers miss the actual top crashes. The core of the problem is that current practice relies on hindsight to identify top crashes, that is, we can accurately identify top crashes only after they have already caused significant trouble for the users.

It should be noted that most of the top crashes do occur in the early phase, although they are not frequent. For example, 16 of the top-20 crashes of Firefox 3.5 occurred at least once during the alpha testing (shown in the bottom-right Gantt chart of Fig. 8). This indicates an opportunity for improving current practice (see Section 6.7 for more discussion on this topic).

3.3 How Can Prediction Improve the Current Practice?

To address this problem of current practice, we advocate a prediction-based approach that does not rely on hindsight to identify top crashes. With our approach, it becomes feasible to identify top crashes during prerelease testing (i.e., alpha or beta testing), and also to react as soon as the first crash reports are received. Rather than waiting for a number of crashes to occur, developers can identify and address the most pressing problems without delay.

To see the benefit of our approach, let us assume that we have an "ideal top-crashes predictor" that can accurately

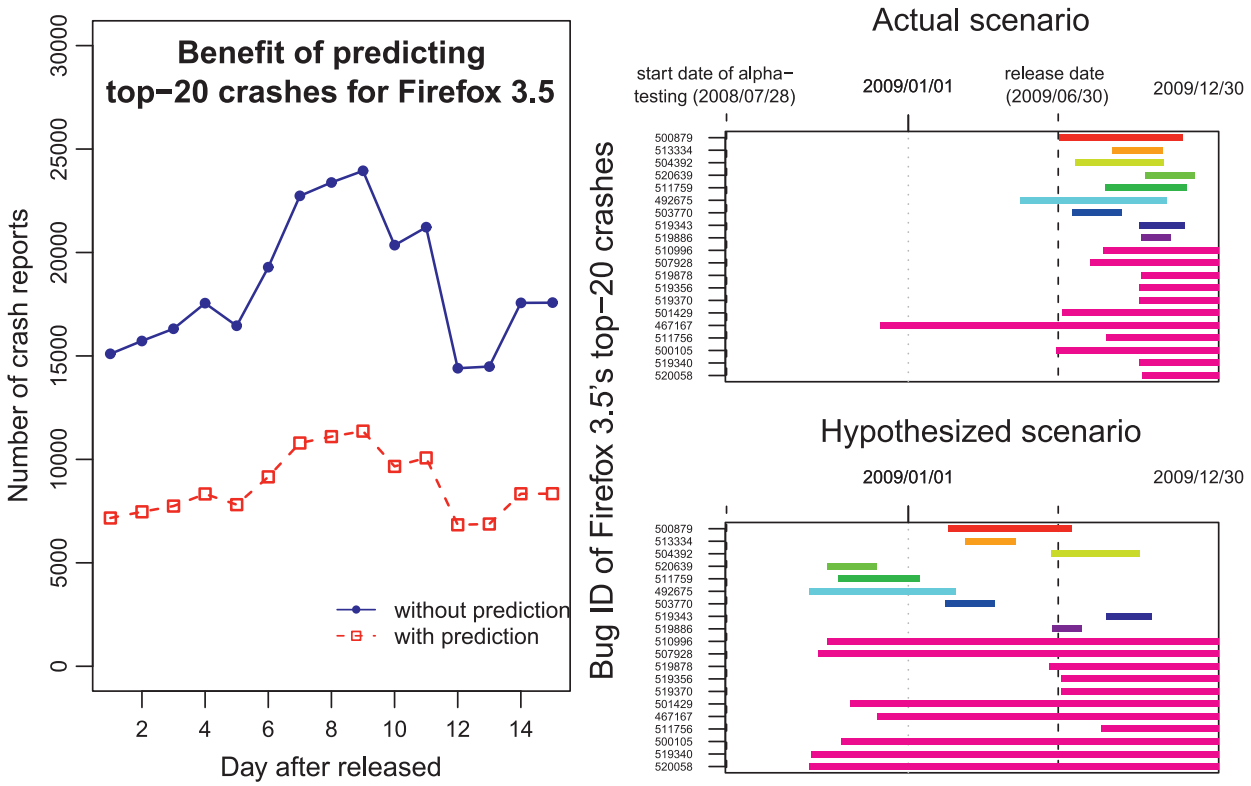


Fig. 8. Comparison between the real situation (no prediction) and the hypothesized scenario (with prediction), in which developers had used top-crash predictor to prioritize their debugging effort for Firefox 3.5.

determine whether a crash is a top-k crash the first time this crash is reported.[3]

Fig. 8 compares the results obtained using the current practice with the results obtained by applying our predictor to prioritize top crashes. In this figure, the curve "without prediction" represents the actual situation. It shows the number of crash reports submitted each day after Firefox 3.5 was released. (These numbers were gathered from the Mozilla crash repository.) The curve labeled "with prediction" represents the hypothesized scenario in which the availability of the predictor allows the developers to identify top crashes at their first occurrence. Specifically, we assume that developers focus on top-20 crashes: Whenever a new crash is reported, they will use the ideal predictor to determine whether it is a top-20 crash.

The two Gantt charts on the right-hand side of Fig. 8 give details on how these two curves are derived. The upper Gantt chart shows the actual scenario. For each top-20 crash, the bar starts with the date when developers start to work on a particular crash and ends with the date when this crash is fixed. To determine these two dates, we queried the bug report corresponding to this crash in the Bugzilla database of Firefox. The lower Gantt chart shows the hypothesized scenario, in which developers apply our ideal predictor on every incoming crash report to determine whether it is a top-20 crash. If the predictor returns a positive answer, developers immediately start working on this crash. To get a fair comparison, we assume that the time required for developers to fix a crash is the same in the actual and hypothesized scenarios. Note that 11 out of 20 top crashes had not yet been fixed at the time when we submitted this paper (29 November 2009), so we kept these 11 crashes as being unfixed in the hypothesized scenario. They are shown as the bottom 11 bars in both Gantt charts.

Fig. 8 shows that a significant amount of crash occurrences can be avoided with a prediction-based approach; overall, the number of crash reports can be reduced by 36.26 percent. In reality, none of the top-20 crashes had been fixed before Firefox 3.5 was released. Had a top-crash predictor been deployed, at least five of the top-20 crashes would have been fixed before release. This result is encouraging because it shows that the prediction-based approach holds promise in improving the current practice. Motivated by this improvement, we propose our approach toward top-crash prediction, which is introduced in the next section.

4 Our Approach

4.1 Overview

To predict the top crashes, we used a machine learning approach, with previous top and bottom crashes as training set. Fig. 2 depicts the overview of our approach.

The first step is to extract top and bottom crashes from Mozilla crash reporting systems (Socorro [54]), as shown in Fig. 2a. These crash reports are from previous versions. We classified them on the basis of the number of crashes. From the extracted crash reports, we obtained crash stack traces and methods in the traces.

The second step is to generate a corpus by analyzing the extracted methods and stack traces, as shown in Fig. 2b. We characterized the crash methods using three groups of features: *history*, *complexity metrics*, and *social network*

3. The value of k is determined by the product manager based on the maintenance budget.

TABLE 1
Set of History Features

Feature name	Description
Top count	How many times the method appears in top traces of the previous versions.
Bottom count	How many times the method appears in bottom traces of the previous versions.
Integrated count	(Top count) − (Bottom count)
Normalized top count	Top count normalized by the maximum top count.
Normalized bottom count	Top count normalized by the maximum bottom count.
Normalized Integrated count	Integrated count normalized by the absolute maximum integrated count.
Top class method	Does the method only appear in top traces? (0 or 1)
Bottom class method	Does the method only appear in bottom traces? (0 or 1)
Both class method	Does the method appear in both top and bottom traces? (0 or 1)
# of crash occurrence	How many times the crash that the method belongs occurred. If the method belongs to several crashes, then the value is set to the maximum number.

analysis features (Section 4.2). Then, we transformed the feature vectors of each method to (stack) trace-based feature vectors since our approach is a stack-trace-based prediction (Section 4.3). From the trace-based feature vectors, we generated a corpus to train a machine learner.

Fig. 2c describes the prediction step. If there is a new crash report, we characterize the stack trace in the report using the three groups of features and feed the feature vector to the trained machine learner. The machine learner predicts whether the crash is at the top or the bottom.

4.2 Features

In this section, we describe the method-level features: history, CM, and SNA. To characterize stack traces in the top and bottom crashes, we consider several sets of features. First, we extracted individual method feature data from crash reports of previous versions (history features) and source code repositories (CM and SNA features). Then, we applied the feature data to each method and generated method-based feature vectors.

4.2.1 History Features—"Methods (Included in Stack Traces) in Top Crashes May Appear in Other Top Crashes Again"

We hypothesized that methods included in top crashes would appear in other top crashes again. Since different versions of the same software product usually have similar architecture and structure, we assumed that methods in the top stack traces reported in the early versions would frequently crash again in subsequent versions.

This assumption was inspired by bug localization research [32], [40]. It is a common understanding that bug occurrence is local. For example, if a method introduced a bug recently, it would soon introduce other bugs [32]. Similarly, we believe our history features reflected these characteristics for the crash methods in stack traces.

Table 1 lists 10 history features used in our approach. The "* count" features represent the occurrences of methods in stack traces. The "normalized * count" features are normalized versions of the "* count" features. They were normalized on the basis of the maximum value of each "* count" feature. "* class method" encodes methods to appear only in the top stack traces, only in the bottom stack traces, or in both stack traces. The feature "# of crash occurrence" represents the number of crash reports that the corresponding methods belong to.

4.2.2 Complexity Metrics Features—"Complex Methods May Crash Often"

We assumed that complex methods crash more frequently than simple methods. Therefore, we characterized methods using complexity metrics features such as the lines of code and number of paths. The choice of these features was inspired by Buse and Weimer's work [11] on path execution frequency prediction. According to their work, complexity metrics features have a strong correlation with "hot paths."

We extracted 28 CM features using Understand C++ [55] from the source code, as shown in Table 2. The features included the number of lines, statement line count, number of parameters, cyclomatic complexity [41], knots (overlapping loop) [57], and maximum nesting. Object-oriented metrics were not taken into account since large parts of our subject programs are not written in object-oriented languages.

TABLE 2
Selected Complexity Metrics Features

Feature name	Description
# of parameters	How many parameters the method gets as input.
# of lines	How many lines the method has (including blank, inactive regions, and comments).
Lines of code	How many lines the method has (including preprocessor).
statement count	How many statements the method has. Note that a line can contain several statements.
cyclomatic complexity [41]	The number of linearly independent cycles.
knots [57]	Measure of overlapping jumps.
maximum nesting	Maximum nesting level of control constructs in the method
ratio of # of comment lines to # of code lines	ratio of the number of comment lines to the number of code lines in the method.

TABLE 3
Set of Social Network Analysis Features

Feature name	Description
In degree	The number of callers to the method.
Out degree	The number of callees to the method.
Betweenness Centrality [26]	Among all shortest paths (i.e. there is no other path that has a shorter length and the same start/end nodes) in the call graph, how many of them pass through the method.
Closeness Centrality [26]	The sum of the lengths of the shortest call paths from the method to all other methods in the call graph, and to the method from all other methods in the call graph.
Page Rank [13]	The fraction of time spent "visiting" the method (measured over all time) in a random walk on the call graph (following outgoing edges from each method).

4.2.3 *Social Network Analysis Features—"Well-Connected Methods Will Be Executed Often and Thus May Crash Often"*

While complexity features capture the internal properties of each method, social network analysis features [26] capture the external properties of a method, i.e., how the method is related to other methods. We considered the most common type of relation: method calls. Our basic observation was that a crash is likely to be a top crash if it includes many well-connected methods that frequently occur in the stack traces of a program. This observation motivated the inclusion of SNA features in our prediction model.

In social network analysis, how well a node is connected (method in our case) can be measured by many different metrics. In our work, we considered five, which are shown in Table 3. Note that these metrics are commonly used by other software engineering researchers (e.g., [39], [42], [59]). To extract these SNA features, we used CodeViz [14] to construct call graphs from the source code of Firefox and Thunderbird, as well as work by Jung [30], to derive the values of these features for each method in the call graph.

4.3 Prediction Model

Our prediction model is based on stack traces, as shown in Fig. 2. For example, suppose that an unknown stack trace t_i is given. Our model predicts if t_1 belongs to the top or bottom crashes. Therefore, we used trace-based feature vectors rather than method-based feature vectors.

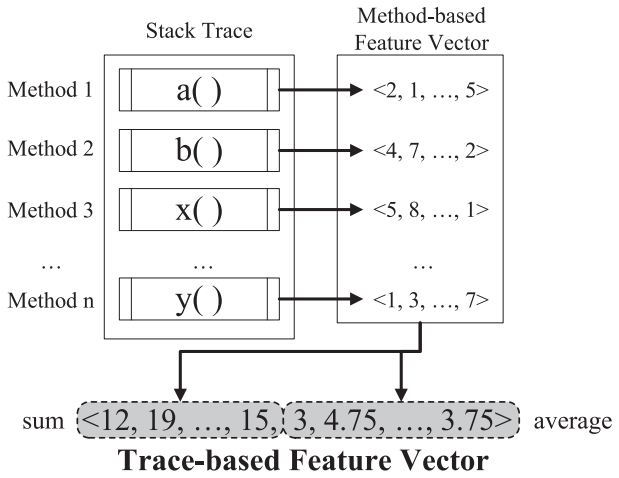


Fig. 9. Trace-based feature vector transformation from method-based feature vectors. For each method in a stack trace, method-based feature vectors are generated. Method-based feature vectors are transformed into a trace-based feature vector. This vector has the sum and average values of each element in the method-based feature vectors.

To obtain trace-based feature vectors, we integrated the method-based feature vectors as shown in Fig. 9. For example, suppose that a crash has a crashing stack trace that includes the methods: $[\mathrm{a}, \mathrm{b}, \mathrm{x}, \ldots, \mathrm{y}]$. These methods have individual feature vectors such as $a = \langle 2, 1, \ldots, 5 \rangle$, $b = \langle 4, 7, \ldots, 2 \rangle$, $x = \langle 5, 8, \ldots, 1 \rangle$, and $y = \langle 1, 3, \ldots, \rangle$. Then, our approach obtained the sum of each ith element of the method-based feature vectors. In addition to the sum values, we obtained the average values of each element. Since a method-based feature vector has 43 elements (10 history, 28 CM, and five SNA features), a trace-based feature vector has 86 features (43 sum and 43 average features).

As an example of such features, consider two feature vectors of `CloseRowObject()` (a top crash) and `GetDirectoryFromLB()` (a bottom crash) where these two crashes are from Thunderbird 3.0. In the feature vector of the top crash, the averages of "In degree" and "Out degree" features were 5 and 1.82, respectively (the sum values are 115 and 42), while, in the vector of the bottom crash, those values were 0.2 and 0.34 (the sum values are 7 and 12). "Betweenness centrality" values (average) of the two crashes are 1.83 (top) and 0.34 (bottom), respectively. The top crash is connected to more methods than the bottom crash and is placed on more shortest paths of call graphs. This supports our hypothesis that a well-connected crash is more likely to be a top crash.

In the case of the history feature group, these examples also support our hypothesis. The average and sum values of the "Top count" feature for the top crash were 30.57 and 703, respectively, while those of the bottom crash were 7.6 and 266. The top crash has more methods that occurred in other top crashes than the bottom crash. This supports our hypothesis that methods in top crashes may appear in other top crashes again.

In the case of the "Cyclomatic" feature, which is one of the complexity metrics feature group, the top and bottom crashes had 70 and 14, respectively. (These are the sum values. The average values were 3.04 and 0.4, respectively.) The top crash has more linearly independent paths than the bottom crash. This supports our hypothesis that complex methods may crash often.

These examples describe how features characterize top and bottom crashes. Although these examples may not perfectly describe the characteristics of top and bottom

TABLE 4
Data Set Used in Our Experiments

Project	# of instances (traces)	# of top crash traces	# of bottom crash traces	# of features		
				History	CM	SNA
Firefox 3.0.9	1236	618	618	10	28	5
Firefox 3.0.10	1632	816	816	10	28	5
Thunderbird 3.0a1	596	298	298	10	28	5
Thunderbird 3.0a2	590	295	295	10	28	5

crashes, they motivate us to investigate on three feature groups.

5 Evaluation

We present the experimental evaluation of our approach in this section. Five research questions will be evaluated:

- RQ1: Is history information indicative of top crashes?
- RQ2: Is the complexity of a method indicative of its chance of triggering top crashes?
- RQ3: Does the connectedness of a method correlate with its chance of occurring in top crashes?
- RQ4: Is the size of training data relevant to the accuracy of top-crash prediction?
- RQ5: Which feature is more indicative than the other features?

This section describes the experiment setup to evaluate our research questions and reports the experimental results.

5.1 Experiment Setup

For our experiments, we used real crash reports from two open source systems: Firefox and Thunderbird. To demonstrate the effectiveness of our approach toward unknown stack traces, we explicitly separated the training set and the testing set. For example, we collected a training set from Firefox 3.0.9 and a testing set from Firefox 3.0.10. Sometimes, crashes may not be fixed in the following versions. For example, the crash "`_PR_MD_SEND`" in Firefox 3.0.9 was not fixed in Firefox 3.0.10. As a result, we find that some crashes are reported across different software versions. For fair experiments, we ensured that the reports of the same crash did appear in both the training and the testing sets by removing these reports from our experiments.

Table 4 describes the data sets (corpus) used in our experiments. We collected crash reports for four programs (two versions of Firefox and two versions of Thunderbird). The two Firefox projects had more than 1,000 data instances (i.e., trace-based feature vectors) extracted from the stack trace database, while the two Thunderbird projects had around 590 data instances. Each project had the same number of top and bottom crashes. Each instance was characterized by 10 history, 28 CM, and five SNA features, as described in Section 4.2, and had 86 elements (sum and average of features), as described in Section 4.3.

Specifically, we created training sets as follows:

1. Sort crashes and choose top-20 crashes.
2. Randomly select n (e.g., 40 in the case of Firefox 3.0.9) stack traces for each crash.
3. Choose bottom-20 crashes and select all traces as these crashes had less than 10 crash reports (sometimes only one).
4. Select the additional bottom $20 + k$ crashes and select all traces until the number of traces is equal to the number of top traces. The testing sets were also created in the same manner.

We only used history information in the training set to create our testing set, as we assumed that we did not know the history information of the testing set. For example, we counted how many times the method appeared in top crashes for the training set. It is possible that some methods in the testing set did not appear in the training set. In this case, we set the corresponding history features as missing values [37].

For a machine learner, we used two machine learning algorithms, Naive Bayes (NB) [45] and multilayer perceptron (MLP) [52]. Naive Bayes is a simple probabilistic classification algorithm based on Bayes' theorem [6] with strong naive independence assumptions. It takes training data and calculates probabilities from them. When a new instance is presented, it predicts the target value of the new instance. It is adopted for our evaluation because of its simple structure and fast learning.

MLP is a feedforward artificial neural network [27]. It has several layers of perceptrons, which are simple binary classifiers. Learning in MLP occurs by changing connection weights between perceptrons after the training data are processed. MLP was chosen for our evaluation because MLP can efficiently classify nonlinear problems [52] (we assumed that it is difficult to learn features in trace-based feature vectors using linear functions).

In addition, we applied the feature selection algorithm proposed by Shivaji et al. [53], which is based on a backward wrapped feature selection technique [47]. First, we put features in order according to their predictive power as measured by the information gain ratio [34], a well-known measure of the amount by which a given feature contributes information to a classification decision. Then, we removed the least significant feature from the feature set and measured the top/bottom crash prediction accuracy. Next, we continually removed the next weakest feature and measured the accuracy until there was only one feature left in the feature set. After this iteration, it was possible to identify the best prediction accuracy and the feature set that yielded the best accuracy.

Although our application scenarios consider prediction at an early stage (e.g., alpha or beta-testing phases), our evaluation concerns two subsequent official release versions (Firefox) because we focused on a performance comparison between our approach and the wait-and-see approach. In

TABLE 5
Prediction Results

Subject	Machine Learner	Accuracy	Top crashes			Bottom crashes		
			Precision	Recall	F-Score	Precision	Recall	F-Score
Firefox 3.0.10 (training on Firefox 3.0.9)	NB	62.75	57.3	92.3	70.7	82.6	34.6	48.7
	NB w/ FS	67.52	62.6	83.2	71.4	76.7	52.6	62.4
	MLP	77.26	71.2	89.6	79.4	86.8	65.6	74.7
	MLP w/ FS	80.75	74.7	91.5	82.3	89.7	70.6	79.0
Firefox 3.0.10 (training on Thunderbird 3.0a1)	NB	66.24	59.3	98.1	73.9	95.2	35.9	52.1
	NB w/ FS	69.36	65.0	80.7	72.0	76.1	58.6	66.2
	MLP	71.26	72.9	65.5	66.0	70.0	76.8	73.2
	MLP w/ FS	73.89	71.9	76.3	74.0	76.0	71.7	73.8
Thunderbird 3.0a2 (training on Thunderbird 3.0a1)	NB	64.66	72.1	51.4	60.0	60.3	78.8	68.4
	NB w/ FS	68.26	79.4	47.1	59.1	63.7	88.4	74.0
	MLP	72.44	87.8	54.1	66.9	65.3	92.0	76.4
	MLP w/ FS	74.91	89.5	58.2	70.5	67.6	92.7	78.2
Thunderbird 3.0a2 (training on Firefox 3.0.9)	NB	54.42	69.8	20.5	31.7	51.7	90.5	65.8
	NB w/ FS	62.9	63.9	64.4	64.2	61.8	61.3	61.5
	MLP	54.4	54.0	79.5	64.3	55.9	27.7	37.1
	MLP w/ FS	68.90	63.8	91.8	75.3	83.6	44.5	58.1

Experiments were conducted for four subjects: two same-project subjects and two cross-project subjects. For each subject, Naive Bayes, NB with feature selection, multilayer perceptron, and MLP with FS were used to classify top and bottom crashes. Four criteria were measured: accuracy, precision, recall, *and* F-score. *In terms of accuracy, MLP outperformed Naive Bayes except for the fourth subject, and MLP with FS outperformed MLP and Naive Bayes for all subjects.*

other words, we cannot compare the performance if we predict the alpha version crash stack traces as stated in background (Section 3); the wait-and-see approach does not work for the alpha version. Note that stack traces of alpha versions are the same as those of official versions. Therefore, our evaluation deals with correct subjects.

In the case of Thunderbird, we adopted two subsequent alpha versions for our evaluation because these versions are quasi-official versions, which consist of sufficient crash reports. In addition, crash reports of the latest official version (Thunderbird 2.0) are currently not available. Therefore, no crash report of the version can be collected.

To implement all the machine learning algorithms mentioned above, we used the Weka [56] library.

5.2 Evaluation Measures

Applying a machine learner to a top-crash prediction problem can result in four possible outcomes:

1. predicting a top stack trace as a top stack trace (T → T),
2. predicting a top stack trace as a bottom stack trace (T → B),
3. predicting a bottom stack trace as a top stack trace (B → T), and
4. predicting a bottom stack trace as a bottom stack trace (B → B).

Items 1 and 4 are correct predictions, while the others are incorrect.

We used the above outcomes to evaluate the classification with the following four measures [3], [31], [48]:

- **Accuracy:** the number of correctly classified stack traces divided by the total number of traces. This is a good overall measure of classification performance.

$$\text{Accuracy} = \frac{N_{T\to T} + N_{B\to B}}{N_{T\to T} + N_{T\to B} + N_{B\to T} + N_{B\to B}}. \quad (1)$$

- **Precision:** the number of stack traces correctly classified as expected class ($N_{T\to T}$ or $N_{B\to B}$) over the number of all methods classified as top or bottom stack traces ($N_{T\to T} + N_{B\to T}$ or $N_{B\to B} + N_{T\to B}$).

$$\text{Precision of Top crashed traces } P(T) = \frac{N_{T\to T}}{N_{T\to T} + N_{B\to T}}, \quad (2)$$

$$\text{Precision of Bottom crashed traces } P(B) = \frac{N_{B\to B}}{N_{B\to B} + N_{T\to B}}. \quad (3)$$

- **Recall:** the number of traces correctly classified as top or bottom traces ($N_{T\to T}$ or $N_{B\to B}$) over the number of actual top or bottom stack traces.

$$\text{Top traces recall } R(T) = \frac{N_{T\to T}}{N_{T\to T} + N_{T\to B}}, \quad (4)$$

$$\text{Bottom traces recall } R(B) = \frac{N_{B\to B}}{N_{B\to B} + N_{B\to T}}. \quad (5)$$

- **F-score:** a composite measure of precision $P(*)$ and recall $R(*)$ for each class (top and bottom).

$$\text{F score } F(*) = \frac{2 \times P(*) \times R(*)}{P(*) + R(*)}. \quad (6)$$

5.3 Prediction Results

This section reports our prediction results. First, we applied our approach to two subsequent versions. For example, we trained a model with Firefox and then applied the model to a subsequent version of Firefox. Second, we applied our approach for cross projects. We trained a model on Firefox and applied it to Thunderbird and vice versa. Table 5 shows

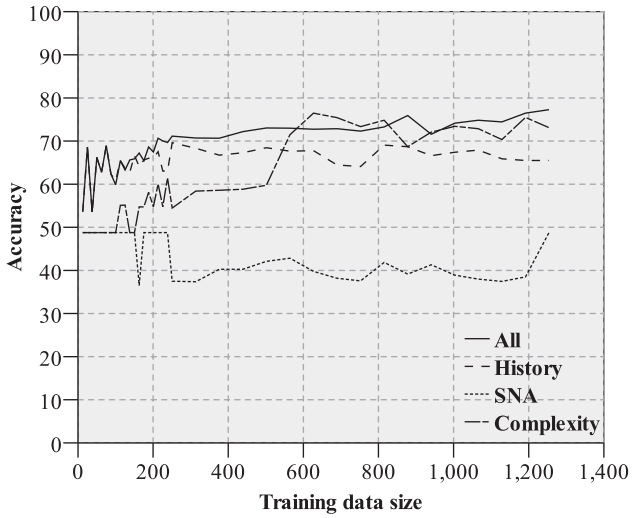


Fig. 10. Prediction accuracy using various training data sizes (Firefox 3.0.10 training on Firefox 3.0.9). This graph shows the accuracy on the basis of different feature groups: social network analysis, complexity metrics, history, and all. At the beginning, the accuracy jitters, but it is stabilized after 250 training instances.

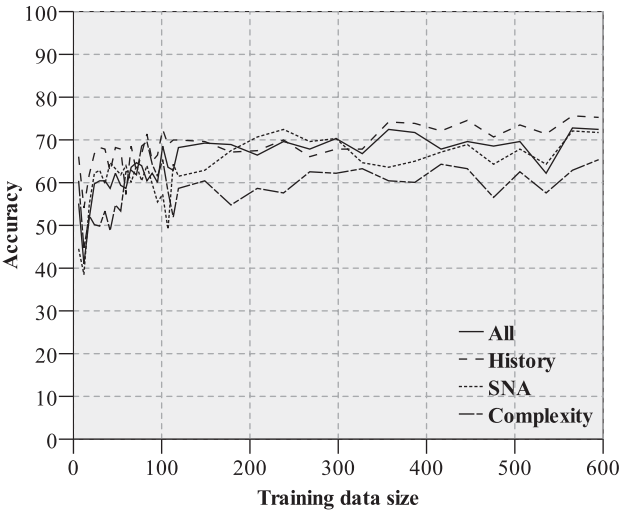


Fig. 11. Prediction accuracy using various training data sizes (Thunderbird 3.0a2 training on Thunderbird 3.0a1). This graph shows accuracy on the basis of different feature groups, the same as Fig. 10. This also has some jitters, but the accuracy stabilized after 150 training instances. Compared to Fig. 10, the accuracy for all four feature groups increased gradually.

the overall results. These results may answer RQ1, 2, and 3. For more details (i.e., predictive power of individual feature groups), see Section 5.5.

For the subsequent versions prediction, our approach predicted top or bottom crashes with > 75 percent accuracy, which is sufficiently high to be useful in practice. Note that the accuracy of a random guess would be around 50 percent since our testing sets were evenly distributed, as shown in Table 5. In terms of top-crash precision, the accuracy of our model was around 90 percent for Thunderbird and 75 percent for Firefox. Overall, we believe our approach is effective and accurate at identifying top crashes as soon as a new crash report arrives.

For the cross-project prediction, the accuracy was around 70 percent, which is slightly lower than that of the subsequent version prediction. However, an accuracy of 70 percent is still considerably better than that of a random prediction. These results suggest that our trained prediction model can be applied to new projects. For example, suppose that the Mozilla group releases a new product. It is possible to predict the new product's crashes as top or bottom using our prediction model trained from Firefox crashes.

MLP mostly outperformed Naive Bayes. We obtained the best results when we used MLP with feature selection. This implies that using the appropriate combinations of features increased the prediction accuracy. We discuss the predictive power of various training data sizes (Section 5.4), and for each feature and feature groups in Section 5.5.

> In our evaluation, models from previous releases predicted top crashes with 68~80% accuracy.

5.4 Size of Training Data

In this experiment, we evaluate the impact of training set size to measure the necessary training data size (i.e., the number of crash instances represented in feature vectors described in Section 4.3) for yielding a reasonable prediction accuracy (around 70 percent) [43] (RQ4). We trained our prediction model using various training set sizes and measured the accuracy. Figs. 10 and 11 show the prediction accuracy with various sizes of training data. We also used different feature groups, history, SNA, CM, and all to measure the accuracy.

In the case of Firefox (Fig. 10), the accuracy jittered when our model was trained with less than 200 training data. However, after 250 training data, the results stabilized and reached a reasonable accuracy. Similarly, the accuracy for Thunderbird (Fig. 11) settled after 150 training data.

> An effective prediction does not require a large number of training instances (200~250 training instances are required).

5.5 Feature Sensitivity Analysis

In this section, we measure and discuss the sensitivity (predictive power) of feature groups and individual features (RQ1, 2, 3, and 5).

To measure the predictive power of each feature group, we trained our prediction model with three different feature groups: history, CM, and SNA (as described in Section 4.2; these feature groups had 10, 28, and five features, respectively). The results are shown in Figs. 10 and 11.

In the case of Firefox, CM features outperformed the other feature groups. They were more than 70 percent accurate and close to the accuracy of all features (for some training data sizes, they even outperformed the accuracy of all features together). The history feature group showed around 65 percent accuracy after 200 training instances. However, the SNA feature group performed worse than random guess.

In the case of Thunderbird, all three types of feature groups showed more than 60 percent accuracy, and the history and SNA feature groups showed more than 70 percent accuracy after 600 training instances. The history feature group even outperformed the case in which all features were used.

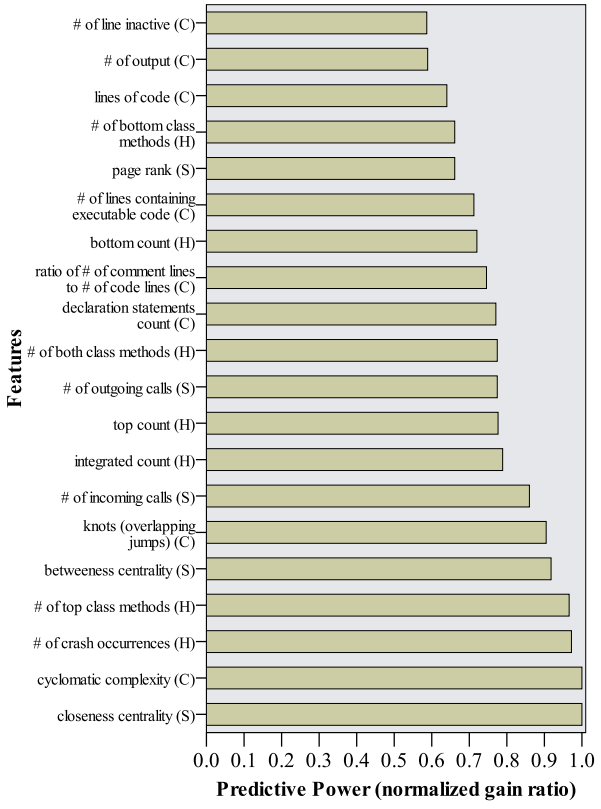


Fig. 12. Normalized predictive power of features established with information gain analysis (Firefox 3.0.9). Closeness and cyclomatic complexity features had the best information gain ratio values. In addition, two history features—number of crash occurrences and number of top class methods—had high information gain ratio values. This shows that complex methods, central methods, and top-crash methods in the past version are more likely to be top crashes.

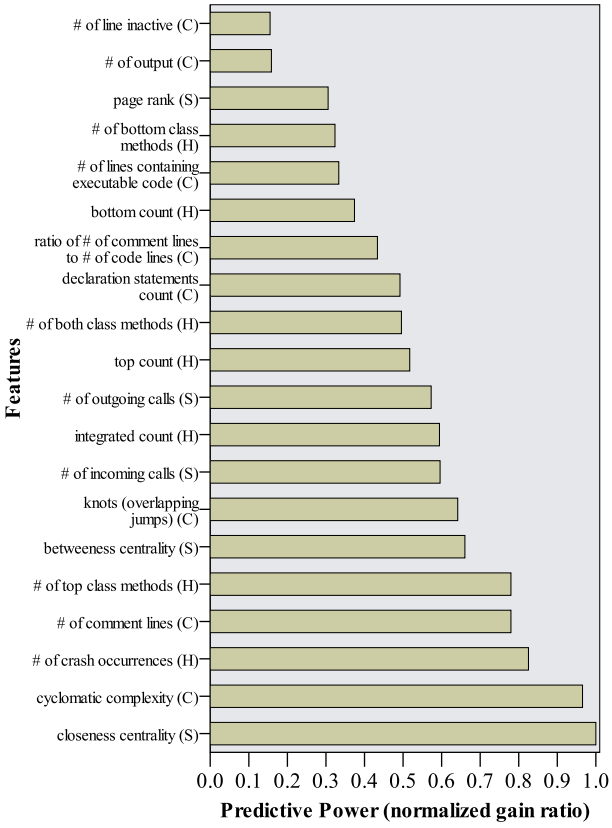


Fig. 13. Normalized predictive power of features established with information gain analysis (Thunderbird 3.0a1). Similarly to the results in Fig. 12, the closeness, cyclomatic complexity, and two history features had the highest information gain ratio values. This also shows that complex methods, central methods, and top-crash methods in the past version were more likely to be top crashes. The number of comment lines also had a high information gain ratio value. This may imply that developers tend to insert more comments in complex methods.

These results show that the three individual feature groups are good predictors for identifying top crashes, as we assumed in Section 4.2. The accuracy of the history feature group indicates that methods in the top stack trace crash again, as we hypothesized. Although this feature group cannot be applied to cross-project prediction, it is useful when a project continually releases subsequent versions and these versions provide new crashes.

The CM features were also good predictors for both projects. Our hypothesis, in which the complexity of methods affects the top-crash prediction, was verified by the result for the complexity features. In addition, these features can be applied to cross-project cases, while history features cannot.

Contrary to our hypothesis, SNA features did not perform well for Firefox. This low accuracy can be interpreted as the fact that two subsequent versions of Firefox do not have much correlation for the prediction of top crashes in terms of SNA features. In spite of this result, SNA features are good predictors, as shown in the Thunderbird case. They can also be applied to cross-project prediction.

For individual features, we measured the information gain ratio [33], [34], [35] of individual features and ranked the features on the basis of the information gain ratio values. The information gain ratio represents how well a feature distinguishes instances. A feature with a high information gain ratio classifies instances more efficiently.

Figs. 12 and 13 show the top-20 features. Firefox and Thunderbird shared many top features. For example, closeness centrality [26] (one of SNA features) had the highest information gain ratio. This verified our assumption in Section 4.2; methods close to the center of call graphs are more likely to be called often and thus be the top crashes. Similar results have also been shown in the literature on bug prediction [59].

Cyclomatic complexity [41] is the second sensitive feature for Firefox and Thunderbird. Cyclomatic complexity represents the number of linearly independent cycles in a method. As we assumed in Section 4.2, this shows that method complexity is an important indicator for determining top and bottom crashes.

In addition to closeness centrality and cyclomatic complexity, two history features—"number of crash occurrences" and "top count"—were highly predictive features. This confirmed that methods in the top-crash traces for the previous version are more likely to reappear in top traces of the subsequent version, as we hypothesized.

> Some features (such as closeness centrality and cyclomatic complexity) are more indicative than other features.

6 Discussion

This section discusses some issues pertaining to our approach and identifies threats to the validity of our experiments.

6.1 High-Profiled Paths

It is possible to profile the path frequency of a program [21] and then predict top crashes using such profile information [11]. We can assume that so-called hot paths crash more frequently than other paths. However, not all hot paths necessarily result in crashes. Instead, we can leverage hot path information to predict top crashes. The SNA features indirectly measure the path frequency. The use of profiling techniques for our prediction model remains as future work.

6.2 Noncrashing Bugs Frequency Prediction

Our approach is not limited to only crashing bugs. In principle, it can be applied in any situation in which 1) the system autonomously detects a failure and 2) a stack trace is available at this point. The failures reported by an automatic problem reporting system are mostly abnormal termination (i.e., a crash), but such a termination is not necessary for our approach if the context information such as stack traces is provided.

6.3 How Many Reports Do We Need to Fix a Crash?

Our approach enables developers to identify and fix top crashes with just a few crash reports. This begs the question: Do a few crash reports contain sufficient information to fix the crash? Obviously, if developers had access to more crash reports, it would be useful. However, similarly to the case of duplicate bug reports [8], most crash reports are duplicates and do not contribute any new information. On the basis of our manual inspections, around 10 to 50 stack traces (i.e., crash reports) for each crash are sufficient for locating and fixing faults.

6.4 Other Approaches to Crash Prioritization

We can suppose that developers use other approaches to crash prioritization. There are three possible approaches: first-come-first-served (FCFS), easier-crash-first, and critical-crash-first. FCFS considers early reported crashes first and is simple to apply. However, early reported crashes do not necessarily indicate that these crashes are more important or frequent than other crashes.

Both easier-crash-first and critical-crash-first approaches are subjective. One developer may consider a specific crash first because he or she has expertise with the module where the crash occurred and thinks it can be readily fixed or it may cause critical consequences. However, these approaches are subjective and do not guarantee a reduction in the number of crash victims because easiness and criticality do not imply that the crash is more likely to occur.

Although some developers use the abovementioned approaches, sometimes some developers may randomly choose crashes to fix. However, they should use our approach if their objective is to reduce the number of crash victims at an early stage.

6.5 Additional Features

The three feature groups that we used might not cover all aspects of crashes. We could add more features to improve the accuracy of top-crash prediction, including crash reason, user comments, and developer's maturity. However, based on our observations, more than 90 percent of crashes are caused by memory access violations and only a very few users (less than 5 percent) submit comments about crashes. In addition, it is difficult to objectively measure a developer's experience with the module where the crash occurs.

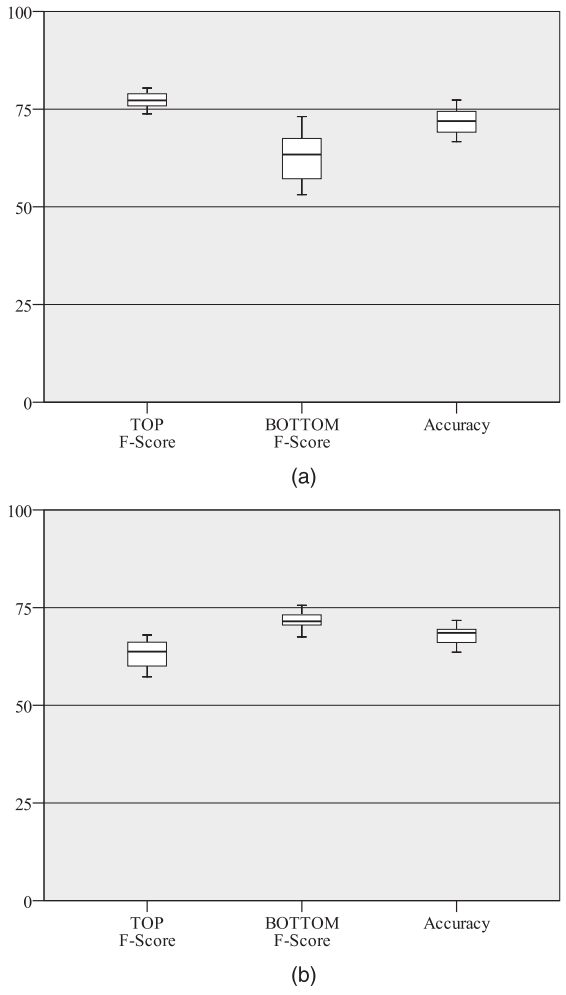


Fig. 14. Variance test results: (a) The variance test result of Firefox. Two hundred fifty training instances are randomly selected and tested on the next version of Firefox. We repeated the same test 20 times. (b) The result of the same test for Thunderbird. Each box plot shows the variance of top/bottom F-score and accuracy.

6.6 Variance Effect

In Section 5.4, we varied the amount of training data to test our technique. The result showed that it requires only a small training set to yield reasonable prediction accuracy. However, a variance effect could be introduced if we (randomly) select training data of a certain size from the entire training set. This variance effect might affect the accuracy of our technique in either a positive or negative way. If this accuracy largely varies due to the variance effect, we might not be able to claim that our technique yields reasonable prediction accuracy. To verify this hypothesis, we conducted a variance test in which 250 training instances were randomly selected and used for training a machine learner (MLP). We tested the learner

TABLE 6
p-Values of the Mann-Whitney Test

Subject	p-value of		
	Top F-Score	Bottom F-Score	Accuracy
Firefox	0.13	0.1	0.08
Thunderbird	0.68	0.4	0.85

using the next version of Firefox and Thunderbird, and repeated 20 tests for each product.

The results of the variance test are shown in Fig. 14. These box plots include the F-score values of top and bottom crashes, and the accuracy for Firefox (Fig. 14a) and Thunderbird (Fig. 14b). In the case of Firefox, the minimum, maximum, mean, and standard deviation values of accuracy are 66.67, 77.33, 71.81, and 3.12, respectively. Those of the F-scores are 73.8, 80.4, 77.28, and 1.90, respectively, for the top crashes, and 53.1, 73.1, 62.81, and 5.8, respectively, for the bottom crashes. In the case of Thunderbird, those of accuracy and F-scores for top and bottom crashes are 63.60, 71.73, 68.0, and 2.33; 57.3 68.0, 63.04, and 3.53; and 67.5, 75.6, 71,25, and 2.03, respectively.

This result implies that our technique might not suffer from the variance effect. Standard deviation values indicate that its average variation was less than 6 percent even when we randomly selected a subset of training instances. We consider that there was no significant difference between the test results.

To statistically verify our claim, we conducted the Mann-Whitney test on the results of each product. Twenty variance test results of each product were divided into two groups: the first 10 results and the remainder. The hypothesis of this test was as follows:

- **Null Hypothesis,** H_0**.** Given the test results, there is **no** significant difference between the two groups in terms of accuracy and F-score.
- **Alternative Hypothesis,** H_1**.** Given the test results, there is a significant difference between two groups in terms of accuracy and F-score.

We calculated the p-values in terms of F-scores and accuracy, as shown in Table 6. In the case of Firefox, the p-values of the top and bottom F-scores and accuracy were 0.13, 0.1, and 0.08, respectively. Those of Thunderbird were 0.68, 0.40, and 0.85, respectively. On the basis of these p-values, we failed to reject the null hypothesis. Therefore, we observed that there was no significant variance effect for either Firefox or Thunderbird.

6.7 Limitations

In the following, we present restrictions and limitations of our technique.

- **The occurrence of top crashes.** To be predicted, a top crash needs to occur at least once in the early product releases. In practice, however, some of the top crashes might only occur after product release. As shown in Section 3, this was the case for four top-20 crashes of Firefox 3.5. For these four crashes, our technique was less helpful.

  There are three main reasons why a crash only occurs after product release. First, the testing effort spent at the early stage is insufficient. This was probably not the case for Mozilla, but it might happen to smaller development teams under strict resource constraints. For them, recent advances in automated testing techniques, such as automated fuzz testing [22], can improve the chance of exposing a top crash at an early stage. Second, the crash can only be triggered in a certain configuration which is not enabled until product release. One of the four Firefox 3.5 top crashes belonged to this case: This crash could not occur in alpha testing because a DEBUG flag masked the crash point. Third, the crash is due to an inadequately tested code patch made before product release. The remaining three Firefox 3.5 top crashes belonged to this case. For two of them, the crash points were exactly located at the lines where developers made hasty changes before release. To avoid such a situation, the development team might introduce a policy mandating a "freeze" period before product release.
- **Training instances.** Because our technique trains on past crash reports, the quantity and quality of these training instances are of concern:
  - First, there might not be a sufficient amount of training data in practice. However, as we illustrate in Section 5.4, an effective prediction does not require a large number of training instances. Our finding was consistent with recent work by Menzies et al. [43], who showed that as few as 50 instances yield as much information as larger training sets. Given this observation, it is likely that the number requirement for training instances can be met, especially if an automatic crash reporting system has been deployed.
  - Second, some of the training instances might not be helpful to top-crash prediction. One main reason might be that they were collected from a very outdated version of the software, which differed significantly from the present version. The use of information in these instances does not contribute to the prediction and sometimes even leads to a worse result. This phenomenon is referred to as "concept drift" by Ekanayake et al. [18]. For this reason, the general guideline of selecting training instances is to prefer those collected from versions closer to the present version. In the future, we plan to conduct further study to investigate concept drift in the context of top-crash prediction.
  - Finally, one particular scenario is that the software is undergoing a first-time release, with no history data available. In this case, it is difficult to apply our technique because it is not a trivial task to create a training set. We are currently investigating the possibility of cross-project prediction.
- **Crash prevention.** The use of some techniques that aim at preventing crashes from happening might potentially limit the applicability of our technique. For example, Demsky and Rinard [17] introduced a technique that automatically repairs inconsistency in

data structures. Michail and Xie [44] introduced another technique for masking GUI errors, particularly crashes. These techniques, however, only temporarily mask the crashes. Ultimately, we still need to fix the underlying cause of the crash. In addition, our technique mainly targets early phases of development, such as alpha and beta testing, in which such crash prevention techniques are rarely deployed.

6.8 Threats to Validity

We identify the following threats to validity:

- **Examined systems may not be representative.** Crash reports for two systems were examined in this paper. These systems might represent the characteristics of a specific group of software systems such as browsers and e-mail clients. However, they might not represent other types of software systems, such data-intensive systems and operating systems.
- **Systems were all open source projects.** All systems examined in this paper were developed as open source projects. Hence, they may not be representative of closed-source development.
- **The information gain ratio measure may be biased.** We used the information gain ratio to measure the feature predictive power and identify important features. However, we only measured individual feature predictive power, while MLP uses a combination of features to build a prediction model. The top individual features shown in Figs. 12 and 13 may have top predictive power for our prediction model. In addition, we compute the information gain ratio using a training set only. However, feature selection based on the information gain ratio yielded the best prediction results, as shown in Table 5.
- **Some stack traces are partially observable.** When a crash occurs, BreakPad [9] collects stack traces per thread and identifies the crashing thread. In the current system, it is impossible to construct an entire stack trace, including the thread spin information. Because of this limitation, we only used the partial stack trace information in the crash thread. However, using the complete stack trace information may yield better prediction results.
- **Training and test instances may be biased.** Although we evenly sampled crash data from the Mozilla crash repository, the possibility exists that data can be biased because some users might not report their crashes due to concerns about security and privacy. However, Firefox and Thunderbird's crash reporting systems [9] force a user to report crashes unless the user terminates the system by killing its process. Moreover, top-crash prediction still provides the benefit of reducing the number of victims for some types of (reported) crashes even if the reports are biased so as to not have other types of crashes such as security and privacy-related crashes.

7 Related Work

In this section, we briefly review research areas related to our approach.

7.1 Crash Reports Analyses

Automatic crash reporting facilities have long been integrated into commercial software (e.g., the Dr. Watson system used by Microsoft [46]). Compared to manually written bug reports [7], crash reports always contain accurate information and require little human effort to submit. However, the analysis of crash reports can be tedious and time consuming if done manually because the raw data they contain (e.g., the heap state and stack frames) are not amenable to human examination. For this reason, researchers have proposed many techniques to automate crash report analysis.

The work most closely related to ours is *failure clustering*, an approach introduced by Podgurski et al. [51]. Their technique applies feature selection, clustering, and multivariate visualization to group failures with similar causes. Such a grouping aids developers in prioritizing debugging efforts because the groups that contain the most failures are likely to be the most significant. Recently, two research groups [28], [38] independently proposed improving clustering accuracy by using automatic fault localization [29]. All of these failure clustering techniques are postmortem analyses, as they suggest what the most common failures in *collected reports* are. In contrast, our technique is a preemptive analysis because it predicts the most common failures in *future reports*.

Another issue in crash report analysis is how to fix the reported crashes. Without tool support, finding the root cause of the reported crashes can be very challenging. *Path reconstruction* techniques have been proposed to address this issue. For example, Liblit and Aiken [36] introduced a technique that supports automatic reconstruction of complete execution paths based on partial execution information such as backtracks and execution profiles. Their technique analyzes control flow graphs and derives a set of plausible paths that agree with available information. Manevich et al. [40] proposed the *PSE* technique, which improved on Liblit's work by incorporating a data flow analysis to reduce infeasible execution paths. These techniques are especially important for debugging predicted top crashes because, at the time of prediction, there are only a small number of crash reports available to developers.

7.2 Frequency Prediction

Frequency prediction refers to the problem of estimating how frequently a program entity (e.g., branch, path, function, or declarative rules) will be exercised in program execution. This is applicable in many domains such as program optimization [24] and reliability assessment [4], [50]. Research in this area can be divided into profile-based [20], program-based [5], and evidence-based [12] approaches.

The profile-based approach samples a few actual executions of the program and generalizes them to predict other executions. It has been successfully applied to predict branch frequency [20]. The profile-based approach requires a workload generator that is capable of simulating the real-world usage of the program. Such a workload generator can be difficult to design.

Unlike the profile-based approach, the program-based approach relies on source code analysis and does not require the generation of a realistic workload. Ball and Larus [5] were among the first to use this approach for

branch frequency prediction purposes. Specifically, they used a loop analysis to predict branches that control the iteration of loops and some simple heuristic rules (e.g., comparing a pointer with NULL usually fails) to predict nonloop branches. One limitation of the program-based approach is that prediction rules have to be derived from experience. However, currently no such rules have been observed for crash stacks.

In [12], Calder et al. proposed the evidence-based approach, which addresses the limitation of the program-based approach while retaining its benefit. The main idea is to first learn prediction rules from a corpus of programs, and then use the learned rules to predict the frequency for new programs. They applied this approach to predict the branch frequency. Recently, Buse and Weimer [11] used a similar approach to predict path frequency. As described in Section 4, our approach shares the same principle as the evidence-based approach. However, we considered predicting the crash frequency, which is different from branch or path frequency. In addition, we introduced the use of social network metrics in our predictions. As shown in Figs. 12 and 13, this contributed to improving the prediction accuracy.

8 Conclusion and Future Work

There is no reason why developers should have to wait for crash reports to pile up before knowing where to start. By learning from past crash reports, we can automatically and effectively predict whether a new crash report is the first of many similar ones to come (and hence deserves immediate attention) or whether it is likely to be an isolated event. By automatically classifying incoming crash reports, developers can quickly fix the most pressing problems, increasing software stability and improving the user's experience. By applying our approach on the crash databases of Firefox and Thunderbird, we found a prediction accuracy of 75-90 percent despite having learned from only a small number of crash reports. This approach is fully automated and easily applicable for any software system where crash data are collected and aggregated in a central database.

Promising results like these call for action on the research side as well. In addition to general optimization and refinement of our approach, our future work will concentrate on the following topics:

- **Integration into the development process.** Predictors like ours would ideally be well integrated into the development process, suggesting and prioritizing actions as soon as the first crash report drops in.
- **Noncrashing problems.** Software failures may not manifest themselves as a crash—the end result may be invalid without the operating or runtime system detecting a problem and, in particular, without a stack trace characterizing the problem. We want to investigate which specific runtime features can be used to predict problem frequency.
- **Avoid concept drift.** The older a crash report, the less relevant it may be for the software at hand. We want to introduce appropriate measures to assign more weight to recent crash reports.
- **Problem topics.** Problem reports also contain user's comments and other natural language information. We want to investigate how specific topics ("printing," "layout," etc.) can be identified and used as additional prediction features.
- **Automatic workarounds.** Once one knows which features are associated with a crash, one could feed back these features to the user, suggesting possible workarounds ("Printing a page with frames has been reported by many not to work properly in this version. Do you want to try printing without frames?")
- **Empirical studies.** Finally, crash report databases offer several opportunities for empirical studies, answering questions like "Which components crash most frequently?," "Which actions are particularly risky?," or "How can we work around the most important crashes?"

More information on our work is available at the project site: http://seapp.sogang.ac.kr/darkrsw/crash.html.

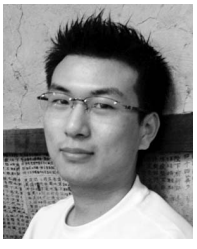
Acknowledgments

This research was supported in part by The Ministry of Knowledge Economy (MKE), Korea, under the Information Technology Research Center (ITRC) support program supervised by the National IT Industry Promotion Agency (NIPA) (NIPA-2010-C1090-1031-0008), in part by the Intelligent Robotics Development Program, one of the Frontier R&D Programs (also funded by MKE, Korea), and in part by the Research Grant Council of Hong Kong (Project No. 612108). The authors would like to thank Christopher Van Der Westhuizen for his helpful ideas on earlier revisions of this paper, and the associate editor and the anonymous reviewers for insightful comments on earlier drafts.

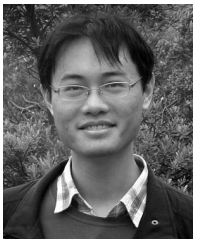
References

[1] A Challenge for Exterminators, http://www.nytimes.com/2006/10/09/technology/09vista.html?_r=2&oref=slogin&pagewanted=print, 2006.
[2] E.N. Adams, "Optimizing Preventive Service of Software Products," *IBM J. Research and Development*, vol. 28, no. 1, pp. 2-14, 1984.
[3] E. Alpaydin, *Introduction to Machine Learning*. MIT Press, 2004.
[4] A. Avritzer, J.P. Ros, and E.J. Weyuker, "Reliability Testing of Rule-Based Systems," *IEEE Software*, vol. 13, no. 5, pp. 76-82, Sept. 1996.
[5] T. Ball and J.R. Larus, "Branch Prediction for Free," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pp. 300-313, 1993.
[6] T. Bayes, "An Essay towards Solving a Problem in the Doctrine of Chances," *Philosophical Trans. Royal Soc. of London*, vol. 53, pp. 370-418, 1763.
[7] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What Makes a Good Bug Report?" *Proc. 16th ACM SIGSOFT Int'l Symp. Foundations of Software Eng.*, pp. 308-318, 2008.
[8] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Duplicate Bug Reports Considered Harmful ... Really?" *Proc. 24th IEEE Int'l Conf. Software Maintenance*, pp. 337-345, Sept./Oct. 2008.
[9] BreakPad, http://code.google.com/p/google-breakpad/, 2009.
[10] Bugzilla@Mozilla, https://bugzilla.mozilla.org/, 2009.
[11] R.P.L. Buse and W. Weimer, "The Road Not Taken: Estimating Path Execution Frequency Statically," *Proc. IEEE 31st Int'l Conf. Software Eng.*, pp. 144-154, 2009.

[12] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn, "Evidence-Based Static Branch Prediction Using Machine Learning," *ACM Trans. Software Eng. and Methodology*, vol. 19, no. 1, pp. 188-222, 1997.
[13] J. Cho, H. Garcia-Molina, and L. Page, "Efficient Crawling through URL Ordering," *Computer Networks and ISDN Systems*, vol. 30, nos. 1-7, pp. 161-172, 1998.
[14] CodeViz, www.skynet.ie/mel/projects/codeviz/, 2009.
[15] Connecting with Customers, http://www.microsoft.com/mscorp/execmail/2002/10-02customers.mspx, 2006.
[16] Crash Reporter (Mac OS X), http://developer.apple.com/technotes/tn2004/tn2123.html, 2009.
[17] B. Demsky and M. Rinard, "Automatic Detection and Repair of Errors in Data Structures," *Proc. 18th Ann. ACM SIGPLAN Conf. Object-Oriented Programing, Systems, Languages, and Applications*, pp. 78-95, 2003.
[18] J. Ekanayake, J. Tappolet, H.C. Gall, and A. Bernstein, "Tracking Concept Drift of Software Projects Using Defect Prediction Quality," *Proc. Sixth IEEE Int'l Working Conf. Mining Software Repositories*, pp. 51-60, 2009.
[19] N.E. Fenton and M. Neil, "Software Metrics: Roadmap," *Proc. Conf. The Future of Software Eng.*, pp. 357-370, 2000.
[20] J.A. Fisher and S.M. Freudenberger, "Predicting Conditional Branch Directions from Previous Runs of a Program," *Proc. Fifth Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 85-95, 1992.
[21] GNU Binutils, http://www.gnu.org/software/binutils/, 2009.
[22] P. Godefroid et al., "Automated Whitebox Fuzz Testing," *Proc. Network Distributed Security Symp.*, 2008.
[23] K. Goseva-Popstojanova and M. Hamill, "Architecture-Based Software Reliability: Why Only a Few Parameters Matter?" *Proc. 31st Ann. Int'l Computer Software and Applications Conf.*, pp. 423-430, 2007.
[24] R. Gupta, E. Mehofer, and Y. Zhang, *Profile-Guided Compiler Optimizations*, pp. 143-174. CRC Press, 2002.
[25] M. Hamill and K. Goseva-Popstojanova, "Common Trends in Software Fault and Failure Data," *IEEE Trans. Software Eng.*, vol. 35, no. 4, pp. 484-496, July/Aug. 2009.
[26] R.A. Hanneman and M. Riddle, *Introduction to Social Network Methods*. Univ. of California, 2005.
[27] J. Hopfield, "Neural Networks and Physical Systems with Emergent Collective Computational Abilities," *Proc. Nat'l Academy of Sciences USA*, vol. 79, pp. 2554-2558, 1982.
[28] J.A. Jones, J.F. Bowring, and M. Harrold, "Debugging in Parallel," *Proc. Int'l Symp. Software Testing and Analysis*, pp. 16-26, 2007.
[29] J.A. Jones and M.J. Harrold, "Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique," *Proc. 20th IEEE/ACM Int'l Conf. Automated Software Eng.*, pp. 273-282, 2005.
[30] Jung, http://jung.sourceforge.net, 2009.
[31] S. Kim, E.J. Whitehead Jr., and Y. Zhang, "Classifying Software Changes: Clean or Buggy?" *IEEE Trans. Software Eng.*, vol. 34, no. 2, pp. 181-196, Mar./Apr. 2008.
[32] S. Kim, T. Zimmermann, E.J. Whitehead Jr., and A. Zeller, "Predicting Faults from Cached History," *Proc. 29th Int'l Conf. Software Eng.*, pp. 489-498, 2007.
[33] S. Kullback, "The Kullback-Leibler Distance," *The Am. Statistician*, vol. 41, pp. 340-341, 1987.
[34] S. Kullback and R.A. Leibler, "On Information and Sufficiency," *The Annals of Math. Statistics*, vol. 22, no. 1, pp. 79-86, 1951.
[35] S. Kullback, *Information Theory and Statistics*. John Wiley and Sons, 1959.
[36] B. Liblit and A. Aiken, "Building a Better Backtrace: Techniques for Postmortem Program Analysis," technical report, Univ. of California, Berkeley, 2002.
[37] R.J.A. Little and D.B. Rubin, *Statistical Analysis with Missing Data*. Wiley, 2002.
[38] C. Liu and J.W. Han, "Failure Proximity: A Fault Localization-Based Approach," *Proc. 14th ACM SIGSOFT Int'l Symp. Foundations of Software Eng.*, pp. 46-56, 2006.
[39] L. Lopez, J.M. Gonzalez-Barahona, and G. Robles, "Applying Social Network Analysis to the Information in CVS Repositories," *Proc. First Int'l Workshop Mining Software Repositories*, 2004.
[40] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang, "PSE: Explaining Program Failures via Postmortem Static Analysis," *Proc. 12th ACM SIGSOFT 12th Int'l Symp. Foundations of Software Eng.*, pp. 63-72, 2004.
[41] T.J. McCabe, "A Complexity Measure," *Proc. Second Int'l Conf. Software Eng.*, p. 407, 1976.
[42] A. Meneely, L. Williams, W. Snipes, and J. Osborne, "Predicting Failures with Developer Networks and Social Network Analysis," *Proc. 16th ACM SIGSOFT Int'l Symp. Foundations of Software Eng.*, pp. 13-23, 2008.
[43] T. Menzies, B. Turhan, A. Bener, G. Gay, B. Cukic, and Y. Jiang, "Implications of Ceiling Effects in Defect Predictors," *Proc. Fourth Int'l Workshop Predictor Models in Software Eng.*, pp. 47-54, 2008.
[44] A. Michail and T. Xie, "Helping Users Avoid Bugs in GUI Applications," *Proc. 27th Int'l Conf. Software Eng.*, pp. 107-116, 2005.
[45] D. Michie, D.J. Spiegelhalter, and C.C. Taylor, *Machine Learning, Neural and Statistical Classification*. Prentice Hall, 1994.
[46] Microsoft Online Crash Analysis, http://oca.microsoft.com/en/dcp20.asp, 2009.
[47] A. Miller, *Subset Selection in Regression*. Chapman & Hall/CRC, 2002.
[48] D. Montgomery, G. Runger, and N. Hubele, *Engineering Statistics*. Wiley, 2001.
[49] Mozilla Crash Report, http://crash-stats.mozilla.com/, 2009.
[50] J.C. Munson and S. Elbaum, "Software Reliability as a Function of User Execution Patterns," *Proc. 32nd Ann. Hawaii Int'l Conf. System Sciences*, vol. 8, p. 8004, 1999.
[51] A. Podgurski, D. Leon, P.A. Francis, W. Masri, M. Minch, J. Sun, and B. Wang, "Automated Support for Classifying Software Failure Reports," *Proc. 25th Int'l Conf. Software Eng.*, pp. 465-475, 2003.
[52] D.E. Rumelhart, G.E. Hinton, and R.J. Williams, "Learning Internal Representation by Error Propagation," *Parallel Distributed Processing: Exploration in the Microstructures of Cognition*, pp. 318-362, MIT Press, 1986.
[53] S. Shivaji, E.J.W. Jr., R. Akella, and S. Kim, "Reducing Features to Improve Classification-Based Bug Prediction," *Proc. 24th IEEE/ACM Int'l Conf. Automated Software Eng.*, Nov. 2009.
[54] Socorro, http://code.google.com/p/socorro/, 2009.
[55] Understand for C++, http://www.scitools.com/products/understand/, 2009.
[56] Weka, http://www.cs.waikato.ac.nz/ml/weka/, 2009.
[57] M.R. Woodward, M.A. Hennell, and D. Hedley, "A Measure of Control Flow Complexity in Program Text," *IEEE Trans. Software Eng.*, vol. 5, no. 1, pp. 45-50, Jan. 1979.
[58] A. Zeller, "Isolating Cause-Effect Chains from Computer Programs," *Proc. 10th ACM SIGSOFT Symp. Foundations of Software Eng.*, pp. 1-10, 2002.
[59] T. Zimmermann and N. Nagappan, "Predicting Defects Using Network Analysis on Dependency Graphs," *Proc. 30th Int'l Conf. Software Eng.*, pp. 531-540, 2008.



Dongsun Kim received the BEng and MS degrees in computer science from Sogang University, Seoul, Korea, in 2003 and 2005, respectively. He is currently working toward the PhD degree at Sogang University. His research interests include dynamic software architecture, self-adaptive software, mining software repositories, and source code analysis. He is a member of the IEEE and the IEEE Computer Society.



Xinming Wang received the BEng degree from Zhongshan University of China in 2002 and the MEng degree from the Chinese Academy of Sciences in 2005. He is currently working toward the PhD degree at the Hong Kong University of Science and Technology. His research interests include software testing and analysis, program debugging, and software mining. He is a student member of the IEEE and the IEEE Computer Society.

Sunghun Kim received the PhD degree from the Computer Science Department at the University of California, Santa Cruz in 2006. He is an assistant professor of computer science at the Hong Kong University of Science and Technology. He was a postdoctoral associate at the Massachusetts Institute of Technology and a member of the Program Analysis Group. He was a chief technical officer (CTO), and led a 25-person team for six years at Nara Vision Co. Ltd., a leading Internet software company in Korea. His core research area is software engineering, focusing on software evolution, program analysis, and empirical studies. He is a member of the IEEE and the IEEE Computer Society.

Andreas Zeller received the PhD degree in computer science from TU Braunschweig, Germany, in 1999, and has served on the Faculty of Saarland University since 2001. He is a professor of software engineering at Saarland University in Saarbrücken, Germany. His research interest lies in the analysis of programs and processes, especially the analysis of why programs fail to work as they should. In 2009, he received the *ACM SIGSOFT Impact Paper Award* for his work on delta debugging as the most influential software engineering paper of 1999. His book *Why Programs Fail* received the *2005 Software Productivity Award* as one of the three most productivity-boosting books of the year. He has served on the editorial boards of the *ACM Transactions on Software Engineering and Methodology* and *Springer Journal on Empirical Software Engineering.* He is a member of the IEEE and the IEEE Computer Society.

S.C. Cheung received the MSc and PhD degrees in computing from Imperial College London. He is a faculty member in the Department of Computer Science and Engineering at the Hong Kong University of Science and Technology. He actively participates in the research communities of software engineering and service-oriented computing. He has served on the editorial board of the *IEEE Transactions on Software Engineering* (*TSE*), and the *Journal of Computer Science and Technology* (*JCST*). His research interests include context-aware computing, service-oriented computing, software testing, fault localization, RFID, and cyber-physical systems. He is a senior member of the IEEE and a member of the IEEE Computer Society.

Sooyong Park received the Bachelor of Engineering degree in computer science from Sogang University, Seoul, in 1986, the Master of Science degree in computer science from Florida State University in 1988, and the PhD degree in information technology with a major in software engineering from George Mason University in 1995. He is a professor in the Computer Science Department at Sogang University and the director of the Requirements and Validation Engineering Center that is supported by the Korean Ministry of Knowledge and Economy. During 1996-1998, he served as a senior software engineer at TRW ISC. He is actively involved in academic activities, including as president of the Korean Software Engineering Society, a steering committee member of the Asian-Pacific Software Engineering Conference, and a guest editor of the *Communications of the ACM* December 2006 issue on software product lines. His research interests include requirements engineering, self-managed software architecture, and software product-line engineering. He is a member of the IEEE and the IEEE Computer Society.