Identifying Recurring Faulty Functions in Field Traces of a Large Industrial Software System

[1]Syed Shariyar Murtaza, [2]Nazim H. Madhavji, *Senior Member; IEEE*, [3]Mechelle Gittens, *Member; IEEE*, [4]Abdelwahab Hamou-Lhadj, *Member; IEEE*

***Abstract*—Software maintainers use the traces of field failures to understand and diagnose faulty functions that cause the system to fail. Despite their usefulness, traces from the field can be quite overwhelming, especially for software systems with a vast client base. In the execution of realistic applications, many of them being millions of lines of code, there are just too many traces that are generated. In addition, traces are known to be extraordinarily large, which further complicates matters. Fortunately, not all field failures are caused by new faults. In fact, previous studies showed that 50% to 90% of field failures are due to previously known faults. In this paper, we propose a machine learning approach that automatically detects recurring faulty functions in the traces of new field failures. We achieve our goal by training decision trees on earlier resolved traces of system failures from the current and prior releases of the system. When applied to a large industrial system with 20 million lines of code and 200,000 functions, our approach was able to detect recurring faulty functions in the traces of field failures with an accuracy of 90%, to even 97% in some cases.**

***Index Terms*—Recurrent faults, software maintenance, crashing failures, non-crashing failures, function call traces, decision tree.**

ACRONYMS AND ABBREVIATIONS

LOC	**Lines of Code**
MDL	Minimum Description Length
WPM	**Windows Performance Monitor**

I. INTRODUCTION

Maintainers use failure reporting techniques to collect information about system failures in the field. System failures in the field can be of two types: crashing failures, and non-crashing failures. Examples of failure reporting tools for crashes include the Windows Error Reporting tool [44], the Mozilla crash reporting system [28], and Ubuntu's Apport crash reporting tool [42] that collect stack traces (function calls on the stack). Examples of tools for non-crashing failures include IBM DB2 [26], and IBM WebSphere [16] reporting systems that collect all of the executed function calls from a specific time[1].

Despite their usefulness, field traces of software products with a large client base can be quite overwhelming to software developers. There are just too many traces that can be reported by the users. This problem is further complicated by the fact that typical traces of failures can be quite large. It is often an arduous task to analyze their content. Fortunately, not all the traces of failures yield new faults. In fact, studies have shown that, when a software product has a large number of instances (copies of the same software product) in the field, approximately 50% to 90% of the failures occur due to previously known faults [51], [19], [3]. We refer to such faults as *recurring faults*. Thus, when a new failure trace is reported, it is most likely due to a fault that was previously reported in other failure traces. The same fault reappears because users do not install patches (updates) on time, or vendors do not provide patches on time [18]. The users do not update systems on time because an updated software application could cause other software applications to crash, and result in a loss of valuable time and money. Similarly, delays from vendors occur because of the time to diagnose and fix the faults. During that time, many faults would reappear in different instances of the deployed software systems.

To aid developers in reducing the time spend analyzing the traces of field failures, and identifying faults that cause the field failures, it is therefore important to automatically identify recurring faults in the traces of field failures. Previous techniques use clustering algorithms to address this problem of diagnosing recurrent faults that cause crashing [3], [8], [10], [19], and non-crashing [33] failures. Usually, these techniques measure a similarity distance between the function call traces of failures, and form groups (clusters) of the traces with minimal distances. Developers then go through the function call traces of a group to discover the locations of faults. Ideally, each group should have one failure type (e.g., crash type), but a group can be associated to more than one failure types.

In this paper, we propose to use supervised learning; more specifically, we generate decision trees from historical traces of field failures with faulty functions already known. We then use decision trees to identify recurring faulty functions in new traces of crashing and non-crashing failures. Our historical collection of failure traces consists of the traces from the

[1]Non-crashing failures can manifest themselves long after the execution of the fault, and are difficult to resolve than crashes.

current and prior releases. The intuition behind our solution is that faulty functions persist across releases [20], and a majority (50 to 90%) of them are recurrent [3], [19], [51]. Unlike clustering based techniques, our approach is capable of pinpointing recurring faulty functions that cause failures, instead of a group of similar faults. In addition, clustering techniques are known to suffer from various limitations including the selection of the adequate distance measure, the clustering algorithm, etc.

We evaluated our approach on field traces of a very large industrial system of 20 million lines of code LOC, 200,000 distinct functions, and a user base of approximately one million users[2]. Our results show that, on average, 90% of the recurring faulty functions in failure traces are correctly identified across the releases. Our results also show that functions can have different faults, but recurring faulty functions can still be accurately identified because of the similarity of traces.

This work is an extension of our previous work where we presented a technique that aims to identify recurring faults using only the current release [29]. We applied that earlier approach to small open source utilities (around 1000 LOC) with small failure traces generated using test suites. This paper particularly contributes by showing that recurring faulty functions, irrespective of the type of faults, can be identified using field traces of different releases too. It also contributes by evaluating the approach on a realistic commercial application with traces collected from the field. It also shows that different types of events (e.g., exception thrown) present in failure traces of commercial applications do not contribute significantly in automatic fault diagnosis, and removal of unnecessary events can reduce the trace size by approximately 50%. In short, we have made many fine adjustments to the technique to make it scalable and applicable in the industrial context.

This paper continues as follows. The next section describes our technique. Section III shows the case study of a large commercial application. Section IV explains threats to validity. Section V describes related techniques. Section VI concludes, and describes future work.

II. The Approach

We term our technique, F007, the faulty function finder. The key objective is to discover faulty functions in system failure traces by using earlier system failure traces of the same release or previous releases. F007 actually builds decision tree models from historical system failure traces. Faulty functions in historical traces are known. F007 then uses the decision tree models to diagnose faulty functions in newly generated traces of system failures from the field. This problem should not be confused with the problem of predicting new faults. As mentioned earlier, our focus is on helping software maintainers reduce the number of field failure traces they need to look at by automatically pinpointing the faults if the failures are caused by previously reported faults.

Our approach encompasses four main steps. The first one is to collect function call traces that are generated when the system fails. In this paper, we include traces of both crashing and non-crashing system failures. In the second step, we train decision trees on the system failure traces of known faulty functions. We actually train one decision tree for each faulty function. We will discuss the rationale behind this approach in the next subsections. The next step is a testing step. Whenever a new system failure trace arrives from the field, we pass it to the decision trees. Each decision tree then associates the trace with its knowledge of earlier traces, and their corresponding faulty functions. Each decision tree then emits the probability of its faulty function for the new trace. The suspected faulty functions from the decision trees are then arranged in a ranked list in decreasing order of their probabilities. The intuition behind this ranking is that the function ranked higher is more likely to be faulty than the function ranked lower. Finally, the list of suspected faulty functions with their ranking is presented to software developers for evaluation. These steps are detailed in the subsequent subsections.

A. Collecting Traces of Failures

In this paper, we focus on traces of function calls because, in practice, function call traces are the commonly collected traces from the field (see Section I). Other traces such as statement-level traces can also be used, but they tend to be not practical because of the additional overhead they cause during trace generation.

A typical scenario for failure analysis in the deployed system is the following [27]. A software service analyst receives a phone call from a customer reporting software failure. The analyst has to quickly determine the root cause of the failure. The analyst tries to determine whether it is a recurring fault, already known from another customer, or it is a new fault. If it is a recurrent fault, then the analyst will quickly provide the customer a fix-patch or another known solution. If it is a new fault, then the analyst has to notify developers in the maintenance team to start detailed investigation about the origin of the new fault. In both cases, the customer must be quickly provided with a solution; the faster the solution, the better the customer satisfaction.

To determine the recurrent fault, the analyst asks the customer to reproduce the failure by enabling a tracer [16], [26]. The analyst collects the trace from the customer. An example of a raw failure trace from the customer is shown in Fig. 1. The trace contains function entry, function exit, function probe point, and error codes (exceptions) for each thread of a process. Additional trace information may also be collected via logging programs for specific operating systems, such as the Windows Performance Monitor (WPM). WPM generates performance counters that monitor characteristics such as CPU state and disk usage, and configuration information such as important values in the Windows registry. Hence, the trace report may also contain separately the profiling information about the software system; e.g., memory usage, CPU usage, etc.

[2]System name is anonymous due to proprietary reasons.

```
Process_id = 1104 Thread_id = 1
  Foo200 entry
  | Foo2340 entry
  | Foo2340 exit
  Foo200 exit

Process_id = 8869 Thread_id = 1
  Foo1 entry
  | Foo2 entry
  | Foo2 probe 10
  | Foo2 mbt [memory allocation probe]
  | Foo2 exit  [Error Code: A1]
  | Foo3 entry
  | Foo3 probe 20
  |  | Foo2 entry
  |  | Foo2 exit [ Error Code: B1]
  | Foo3 exit [Error Code: B1]
  Foo1 exit
  ……..
```

Fig. 1. An example of a function call trace with entry events, exit events, probe points, and error codes.

Once the trace is collected from the customer, the analyst compares it against a library of existing traces collected in the past, whose fault origins are already known. The analyst identifies a set of similar traces in the library by filtering (searching) the library by the names of functions present in the new faulty trace. The filtered subset of the traces from the library is then examined manually to identify common patterns with the new faulty trace. If the analyst finds an existing trace with common patterns, then it is a recurrent fault; else it is a newly discovered fault. Once recurrence is ascertained, then the analyst can determine the fixes from the fault management database for the selected trace in the library. A fault management record for the trace may contain the fault identifier, the problem description, the problem solution, fixes, the status, the open date, the close date, comments, the components affected, the version number, the operating system, and maybe more.

In short, this process is similar to a usage event of an Internet search engine. A user queries the search engine with keywords, and the engine's algorithm returns a list of ranked web pages. The user then examines the returned pages to identify the relevant pages. The better the algorithm of the search engine, the less time the user would spend in examining the web pages. This model also applies to recurrent fault diagnosis. When there are thousands of traces present in the library, then the manual approach becomes daunting. A fault diagnosis technique, like F007, can facilitate the analyst in getting the ranked list of the most relevant fault identifiers, and traces associated with fault identifiers. The analyst then only needs to review the top few traces in the ranked list to determine the recurrent fault, and avoid the laborious work needed for a manual investigation.

Initially, the library of historical failure traces with known faults can be built using field traces, if available; otherwise, in-house traces can be generated from failed test cases. Studies have shown that the origins of in-house and field faults, in many cases, overlap significantly [15]. This library can contain traces (as shown in Fig. 1) of both crashing failures and non-crashing failures (see Section I). F007 trains on these failure traces of known faults by extracting the events of all the processes and threads in a trace. The events include function entry, function probe points, function exit, and error codes, as shown in Fig. 1. The details of training are explained in the next section.

B. Training the Decision Trees on System Failure Traces

We train the decision tree on traces of system failures by transforming the trace information into a dataset, as shown in Table I. A row in Table I **Error! Reference source not found.** represents a trace from a historical collection of failure traces. The columns represent the decision tree attributes, which are the distinct events invoked in the traces (see Fig. 1). For example, Foo1 represents a function, Foo2_P1 represents a function and associated probe point, and Foo1_Err1 represents a function and associated error code. A cell represents the probability of the occurrence of an event E_i in a trace, calculated using (1). The probability of an event $P(E_i)$ is calculated as the ratio of the frequency of an event in a trace $|E_i|$ to the sum of the frequencies of all the events in the trace $\sum_{j=1}^{n} |E_j|$. For example, if an event Foo1 has occurred 50 times in a trace, and a total number of events in a trace is 1000, then the probability of Foo1 is 0.05.

$$P(E_i) = \frac{|E_i|}{\sum_{j=1}^{n} |E_j|} \quad (1)$$

The last column in Table I**Error! Reference source not found.** shows faulty functions for an historical trace. In data mining terminology, faulty functions in the last column are the labels, and the events in the other columns form attributes [45]. In the case of multiple faulty functions, the labels contain the names of multiple faulty functions, shown as the label *Foo3 & Foo6* in Table I**Error! Reference source not found.**. *Foo3 & Foo6* represents that two functions F003 and Foo6 are faulty simultaneously. In a similar manner, we also represent more than two faulty functions.

The reason for selecting single events as attributes in Table I **Error! Reference source not found.** lies in the empirical investigation of our earlier paper [29], where we have empirically investigated that the patterns (sequences) of events do not yield better results than the single events when used with the decision tree. For example, if a trace has four events {E1, E2, E3, E4}, and a decision tree is trained on patterns {E1E2, E2E3, E3E4, etc.} and on only individual events, then the decision tree yields the same accuracy, implying that training on individual events is as efficient as when using patterns. Similarly, we also observed that, if the decision tree

is trained on both single events and patterns (i.e., {E1, E2, E1E2, etc.}), then the results will be the same as training on individual events. Thus, we can avoid using patterns as their extraction causes additional overhead.

Table I

Dataset of functions of traces of a large software system to train the decision trees (for confidentiality reasons, the function names are obfuscated)

	Events in a trace (i.e., Function calls, Function calls with probe point, and function calls with error code)							
	Foo1	Foo2_P1	Foo1_Err1	Foo5	.....	FooN	FooN_Pi	Faulty Functi-ons
Trace 1	.040	.030	.005	.010	.	.050	.010	Foo1
Trace 2	.020	.010	.003	.010	.	.110	.100	Foo5
Trace 3	.001	.005	.010	.003	.	.510	.030	Foo3& Foo6
Trace 4	.023	.001	.002	.040	.	.530	.020	Foo1
……	…..	…..	…..	….	.	….	…	……
Trace n	.004	.032	.111	.003	.	.100	.101	Foo5

(a) Original dataset for all categories

Trace 1	.040	.030	.005	.010	.	.050	.010	Foo1
Trace 4	.023	.001	.002	.040	.	.530	.020	Foo1
Trace 3	.001	.005	.010	.003	.	.510	.030	Others
……	…..	…..	…..	…..	.	…..	…..	……..
Trace n	.004	.032	.111	.003	.	.100	.101	Others

(b) Dataset for function Foo1 against all others

We use the one-against-all approach in training the decision tree classifier [45]. In this approach, a dataset (of traces) with M categories of labels (faulty functions) is decomposed into M new datasets with binary categories. Each new binary dataset Di has a category Ci (where i = 1 to M) labeled as positive, and all other categories are labeled as negative. An example of a dataset of the faulty function Foo1, against all other faulty functions, is shown in part b of Table I **Error! Reference source not found.**. In part b of Table I, all rows are assigned a label of *others* except rows having Foo1 as a label. Similarly, a new dataset is generated for every faulty function in the original dataset.

On each new dataset Di,, the decision tree algorithm is trained, resulting in M trees in total. Empirical evidence shows that training multiple decision trees (one-against-all) on several binary datasets yields better results than training a single decision tree on a dataset with many categories of labels [34].

The decision tree algorithm we used in this paper is C4.5 because of its popularity and tool support [45]. It is also suitable for a dataset with numerical values of attributes, unlike other algorithms such as the ID3 decision tree algorithm which works only with nominal values of attributes [45]. The details of the C4.5 algorithm can be found in the standard textbook by Quinlan [37].

There exist several other classification algorithms such as neural networks, support vector machines, naïve Bayes classifiers, etc. In one of our earlier papers, we have formally compared different classification algorithms on function call traces and system call traces [30]. The classification algorithms include the C4.5 decision tree, Naïve Bayes, Bayesian Belief Network, Multilayer Perceptron (Neural Network), Support Vector Machine, and Hidden Markov Models. We have conducted a Wilcoxon signed rank significance test, and an effect size test. According to the significance test, no significant differences exist among the classification accuracy of classifiers on the function call traces. According to the effect size test, the C4.5 decision tree should be preferred over other classifiers when there are multiple classes. We have also (informally) observed that the C4.5 decision tree is faster in processing time, and can generate rules that can be interpreted by human experts. Therefore, we have chosen the C4.5 decision tree for classification. Nonetheless, other classification algorithms can also be used, and readers are referred to [30] for in-depth results.

We have used one C4.5 classifier with the one-against-all approach. The ensemble methods like Random Forrest, AdaBoost, and Stacking can also be used with the one-against-all approach, instead of only one C4.5 classifier. However, the ensemble classifiers still must be adjusted to generate rankings like our approach does (see Section II.C), and they must be compared with our current approach for improvement in accuracy and time. Thus, it remains outside the scope of this paper which ensemble method is the best for field failure diagnosis.

Moreover, we used Minimum Description Length (MDL) correction and a 25% confidence interval to prune the decision tree. Another alternate would be to use Laplace correction. As noted by Provost and Domingos [36], Laplace correction can improve the accuracy of the decision tree while reducing probability estimation errors. However, we did not find any difference in our results with and without the use of Laplace correction. Thus, we generated the decision tree without the use of Laplace correction.

An excerpt of the C4.5 decision tree when applied to part b of Table I**Error! Reference source not found.** is shown in Fig. 2. Each line contains an event, its probability of occurrence, and the name of the faulty function after a colon sign, if any. The event names represent the tree nodes, and the faulty function name after the colon sign represents the leaf of the tree.

```
Foo3_P4 <= 0.00958333
|  Foo1_Err2 <= 0.0133333: others
|  Foo1 > 0.133333
|  |  Foo5_Err1 <= 0.925926
|  |  |  Foo4<= 0.001
|  |  |  |  Foo2_P1 <= 0.0363636
|  |  |  |  |  Foo6 <= 0.00075: Foo1
|  |  |  |  |  Foo3 > 0.00001: others
……..
……..
```

Fig. 2. An excerpt of the C4.5 decision tree model for the function Foo1 of the large software system.

C. Testing the Decision Trees

Whenever a new failure trace arrives, F007 extracts the same events as the ones used to train the decision trees, and provides the extracted events to the trained decision tree models. Each decision tree, which we trained using the one-against-all approach, predicts its category Ci of the label (i.e., faulty functions in our case) along with the probability of being faulty. The method of generation of the probability from a decision tree can be found in a standard text [37]. In the one-against-all approach, a common method is to select the category Ci with the highest probability as the predicted category (i.e., faulty function) of the trace [45], [34]. We employed the one-against-all approach with a little modification: we ranked the predicted faulty functions in decreasing order of their predicted probabilities. The function list is then presented to the developer with the premise that the higher the function in the list, the more likely it is the faulty function.

An example of a ranked list of faulty functions, predicted by F007, for two different traces obtained from the industrial software system, is shown in Table II. The actual faulty function in the two traces is Foo2, which is ranked at the first position for the first failure trace, and ranked at the second position for the second failure trace.

Table II=

Predicted ranking of functions suspected to be faulty for for new failure traces. (Foo3 & Foo6 is a label representing multiple faulty functions that are predicted to be faulty simultaneously. Foo1, Foo8 are two labels representing two single faulty functions, ranked at the same position, but predicted to be faulty separately. Rest of the labels represent single faulty functions and different rankings.)

	Function	Probability
Trace 1		
Rank 1	Foo2	0.044
Rank 2	Foo1	0.032
Trace 2		
Rank 1	Foo3: Foo6	0.119
Rank 2	Foo2	0.017
Rank 3	Foo1, Foo8	0.010

To accurately evaluate the approach of training and testing the decision trees on our dataset, we actually divided the dataset into three different stratified parts [45]. In the stratification of data, each of the categories of labels (different faulty functions in our case) is represented in approximately the same ratio in each new part as it is in the original dataset. We randomly selected one part (approximately 33% of the dataset) for training the C4.5 decision tree algorithm (using the one-against-all approach), and used the remaining parts (approximately 67%) for testing. The literature recommends selecting more than 50% of the data (more than one part) for training, and the remaining part for testing [45]. However, we used a small proportion of data for training the decision tree because only a limited number of traces are usually available in industry. The training of F007 on a smaller proportion shows that, even by using a limited number of traces, F007 can diagnose faulty functions with good accuracy. Moreover, in the case of the identification of faulty functions across releases, we tested F007 by training it on the traces of earlier releases, and testing it on the latest releases.

III. Case Study on a Large Proprietary Commercial Application

In this section, we show the validation of our approach on a large-scale industrial software product (of size over 20 million lines of code (LOC)) deployed in the field for more than 20 years. The system is written mainly in C and C++. It is developed by several thousands of software engineers over many years, and had many field faults across many functions and components. A component can span many files, and files can encompass many functions. The system has a large user base of millions of users, which makes it a good candidate for the problem of identifying recurring faults in field traces.

A. Collecting System Failure Traces and Executing F007

Table III, first, shows the characteristics of this large industrial application including the number of traces generated from system failures, faulty components, and faulty functions for three releases. The last row in Table III shows the total distinct faulty functions and components across all three releases.

The average size of a trace in Table III is 50 MB, and often the size of a trace reaches few GBs. Due to their large sizes, the traces were not kept in the data repository for a long time, and were purged soon after the resolution of the problem. This purging inhibited us from collecting traces for all the faults. Thus, we collected system failure traces present in the repository for three releases during a period of two years.

For this large industrial application, no explicit records of faulty functions are kept for traces. Instead, a fault identifier is assigned to each failure trace. We have selected all those functions as faulty that were modified by developers due to the fault identifiers. We extracted faulty functions with their scope

(e.g., namespace, file) because two functions in different namespaces can have the same name. We also grouped together all the selected faulty functions of different fault identifiers under one name, if they matched one or more faulty functions of another fault identifier. In Table , the Faulty Functions column shows the number of faulty functions after forming the groups. Similarly, we followed the same procedure for faulty components.

Table III

Characteristics of three releases of the large commercial application

20+ million LOC, 300+ components, approx. 200 K+ functions, average trace size is 50 MB, average number of unique events (function calls, probe points, and error codes) are 10,000 per trace, and 82% recurring faults in failure traces			
	# Failed Traces	# Faulty Components	# Faulty Functions
Release 1	269	52	65
Release 2	337	35	47
Release 3	99	30	31
Total Distinct Faults (Union)		65	103

We implemented F007 using Java, MySQL, and Weka. We executed F007 on a system with 3GB RAM, and a dual core CPU. F007 extracted functions from the traces, and stored them in a MySQL database. The time to parse a trace and store it into the database took up to 10 minutes. This time to process traces was due mostly to MySQL. We used bulk processing techniques to store the trace quickly into MySQL; however, we believe that this time can be further reduced.

Once the traces were stored in the MySQL database, we generated decision trees from them. The time to build multiple decision tree models using the one-against-all approach was approximately 15 minutes. The time to build a single decision tree model without using the one-against-all approach was approximately 8 minutes. The difference in training time between a single model and multiple models is not big. In addition, multiple decision trees using the one-against-all approach yield better classification accuracy [34]. Finally, the new field trace was also processed and stored into the database, and the classification of faulty functions in a new field trace was done instantaneously by the decision trees.

B. Using Different Events for the Identification Faulty Functions

The traces of this large software consist of many events, such as function entry, function exit, function probe points, and error codes (exceptions thrown). Traces are known to be difficult to examine because of their large sizes [13], [14]. A wise classification approach would be to only use the right set of events for the discovery of faulty functions. A large number of events can increase the training time, noise, and memory consumption of the classification model. However, removing necessary events can decrease the accuracy of the classification model. Therefore, we decided to train F007 on three heuristics to identify the right set of events for function call traces.

- **Heuristic A**: Train F007 on all the events, which includes function event, function with probe point event, and function with error code event. The intuition is to use all the events in the function call traces to identify faulty functions.
- **Heuristic B**: Train F007 on only function events. The intuition is that we can identify faulty functions by only using the functions in traces without error codes and probe points.
- **Heuristic C**: Train F007 on functions with higher variations in traces. The intuition is that functions with smaller variances do not contribute much in the classification using the decision trees, and we can identify faulty functions without them.

Table IV

Results of F007 using different heuristics of events selection (SD represents the standard deviation of functions)

# of Funcs Reviewed	**Heur A**	**Heur B**	**Heuristic C**			
			SD > 10	**SD > 100**	**SD > 200**	**SD > 400**
1	52.0	52.4	52.8	56.0	54.6	54.6
2	55.1	52.8	53.3	56.8	56.0	55.5
3	56.8	57.7	58.2	61.3	60.4	59.1
4	62.7	58.6	58.6	62.2	61.3	60.4
5	64	64.8	64.8	64.4	63.5	62.6
6	73.7	67.1	67.1	68.4	68.4	68.0
7	74.6	75.1	75.1	78.6	78.6	78.2
8	74.6	75.5	75.5	79.1	79.1	78.6
All	100	100	100	100	100	100

In the above three heuristics, the event function refers to the function exit event, except for the function with probe point events. This description was also the case in the examples shown in Section II. The reason for selecting only function exit events lies in our earlier paper [29]. In our earlier paper, we discovered that the accuracy of the diagnosis of faulty functions using the decision trees is the same for function entry events, function exit events, and both the function entry and function exit events [29]. That is, we can use function entry or function exit events for training the decision trees, but both of them are not needed. This section actually extends this discovery further by evaluating the use of other events found in commercial software applications for fault discovery.

We employed F007 on the three heuristics in a similar manner to the approach described in Section II. The results are shown in Table IV. Table IV shows the cumulative accuracy on the review of each function from the suspected function list of F007. Table IV demonstrates that faulty functions in approximately 80% of the failure traces were correctly diagnosed using F007 up to the review of the 8th suspected function in its list. After that, F007 did not identify any faulty

function, and the maintainer has to review all the functions.

For Heuristic C, Table IV shows the results for functions with standard deviations greater than 10, 100, 200, and 400. For heuristic C, we trained F007 on functions with standard deviations higher than different threshold values from 10 to 500 with steps of 10. We show the results in Table IV for selected threshold values to avoid cluttering the text. We stopped at the threshold of 400 because, beyond this value, all the functions in some traces were removed, resulting in the removal of those traces from the training dataset. The standard deviation of 400 for a function would seem quite high, but it is quite a small variation because the largest standard deviation of a function was 358945.53 in the traces.

It can be observed from Table IV that the accuracy remains similar between different heuristics. We therefore conducted the Wilcoxon signed rank test to determine if there is any significant difference between different heuristics. We chose alpha to be 0.05. A Wilcoxon signed rank test between Heuristic A and Heuristic B resulted in z=0.420, observations = 9, and a two-tailed p=0.674 > 0.05. Similarly, a Wilcoxon signed rank test with 9 observations between Heuristic B and Heuristic C with SD > 400 yielded z=0.630, and p=0.529 >0.05. In both cases, no significant difference exists among the accuracies of heuristics as p > 0.05. However, a significant difference does exist between the number of events. The number of events extracted using Heuristic A were 17331, using Heuristic B were 10481, and using Heuristic C (SD > 400) were 1892.

Thus, we conclude in this section that, when function-call level execution traces are used, then only function events (i.e., function entry events, or function exit events) with a higher standard deviation than 400 are adequate for discovering faulty functions; even error code events and probe point events can be ignored. (However, error codes may not be discarded to understand the type of fault.) This reduction in events in function call traces can facilitate recording only necessary events in large software systems, thereby significantly reducing the size of the trace by up to 50%. For example, in some cases, we have traces of about 4GB (44 million function-calls); and they were reduced to less than 2 GB by removing function entry events, function with probe point events, and error code events. The savings in trace size is significant, especially when it is not possible to store Gigabytes of traces for a longer period to perform data analytics.

Our three heuristics actually reduce the number of attributes before the development of a model using the decision tree. A variety of attribute selection techniques also exists in the data mining literature, such as information gain attribute evaluator, subset attribute evaluator, principal component transformation, etc. [45]. Some of these techniques (such as subset attribute evaluator) resulted in lowering the accuracy, and some in exceeding the memory limit (such as principal component transformation). The heuristics that we used allowed us to reduce the number of attributes (data size) before loading into memory without affecting the accuracy. Further research on comparing different attribute selection techniques is outside the scope of this paper. The results of all the remaining sections are based on the reduced number of events as identified in this section.

C. *Classifying Faulty Functions in Field Failure Traces of a Release*

In Fig. 3, we show the results of F007 on three releases by using a 33% training set, and a 67% test set. Fig. 3 shows the accuracy of F007 when it is trained on a small percentage of system failure traces of a release, and identifies faulty functions in the remaining traces of the same release. We used a small training set to reflect the availability of only limited historical traces as is common in industry. The horizontal axis represents the percentage of a program that needs to be examined by a developer before getting to the faulty function. It is measured by the number of functions reviewed by a developer up to the faulty functions from the ranked list of F007 divided by the total number of functions. It is defined in (2).The vertical axis measures the cumulative percentage of system failure traces that achieve a score within a segment on the horizontal axis. This metric is an effective way of assessing the results, also adopted by other fault localization techniques [50], [17].

$$\begin{bmatrix} \%\,of\ program \\ to\ review \end{bmatrix} = \frac{Functions\ reviewed\ upto\ the\ faulty\ function}{Total\ functions} * 100 \quad (2)$$

We are aware that this metric does not account for the complexity of each function, which varies from one function to another. We are simply measuring, in percentage, the number of functions that a developer needs to look at before reaching the actual faulty function, proposed by our ranking. For this commercial software, we could not get access to the actual source code to count the number of statements of functions or components or measure complexity using other known metrics. Doing so would have helped in finer-grained evaluation in terms of the number of statements reviewed for each function or component. However, it is also known that maintainers do not review all the statements of every function to identify a fault. They review few relevant statements; and by using their experience, they can determine whether the function is faulty or not.

There were about 200,000 functions in the software application, and on average up to 10,000 distinct functions per trace. In the worst case, a developer should review the total number of functions in a trace. However, we assume that a developer would consider a minimum of at least 1,000 functions in total to diagnose faulty functions. Therefore, in all the graphs shown next, we have used 1,000 total functions for(2). In Fig. 3, a point (0.8, 70) means that faulty functions in 70% of the failed traces in the test set were correctly diagnosed upon reviewing 0.8% of the program's functions. A line (without markers) in Fig. 3 shows that no classification is made, and a developer has to use 100% of the program to identify faulty functions. This line goes up to the last point (100,100) on the chart, but it is not shown up to 100% on the chart for better visibility of the markers. Mostly, the line represents those traces that have newer faulty functions and

are not found in the training set.

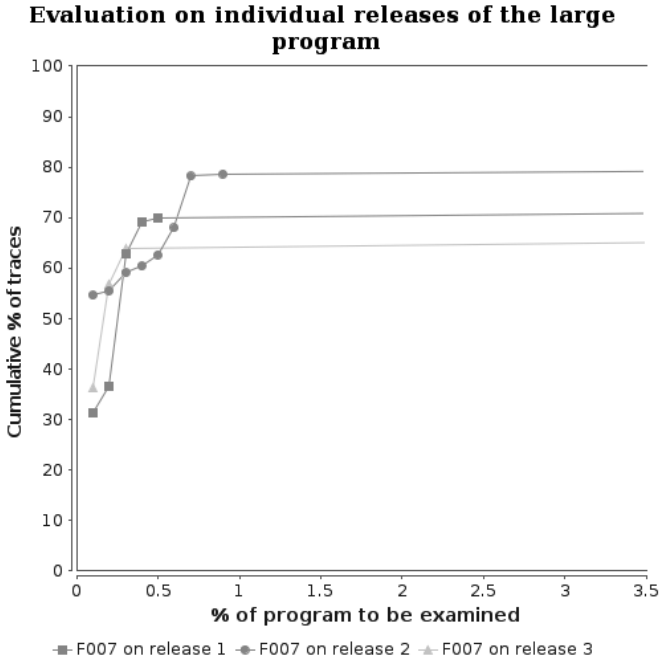

Fig. 3. F007 on three releases of the large commercial application by using 33% of the data for the training set, and 67% of the data for the test set. The horizontal axis is measured by using (2) with the total number of functions being 1,000.

Observe from Fig. 3 that recurring faulty functions in 76% (Release 1), 81% (Release 2), and 66% (Release 3) of the system failure traces are successfully identified by F007 for each of the releases by reviewing less than 0.8% of the program (i.e., 8 functions out of 1,000). In the rest of the cases, some of the faulty functions occurred only once (one trace) in the test set. So these functions were not identified at all by F007 for the sample of traces we used; they were not recurrent, and are represented by the straight line. Recall from Table III that there were 82% recurrent faults in our sample dataset, so the accuracy we obtained (i.e. 65% to 80% depending on the release) is outside the 82% existing recurrent faults. This result is equivalent to an accuracy of 92% (Release 1), 98% (Release 2), and 80% (Release 3) out of 100% recurring faulty functions. The average accuracy is therefore 90% on the review of 0.8% or less of the code (fewer than 8 functions).

We compared our approach against what we call the straw man approach, a random method that a developer can use. In the straw man approach, we generated a ranking of faulty functions from the training set. In this ranking, we ranked the faulty function with the largest number of traces in the first position, the faulty function with the second largest number of traces in the second position, and so on, ending with the faulty function with the smallest number of traces on the last position. We used this ranking instead of the F007 ranking to classify faulty functions in the traces of the test set. The results are shown in Fig. 4. See that by using the straw man approach a developer has to review more code than the F007 to identify faulty functions. The results obtained using F007 are significantly better than the straw man approach. This result also shows that the decision trees are trained well, and their results are better than a random straw man approach.

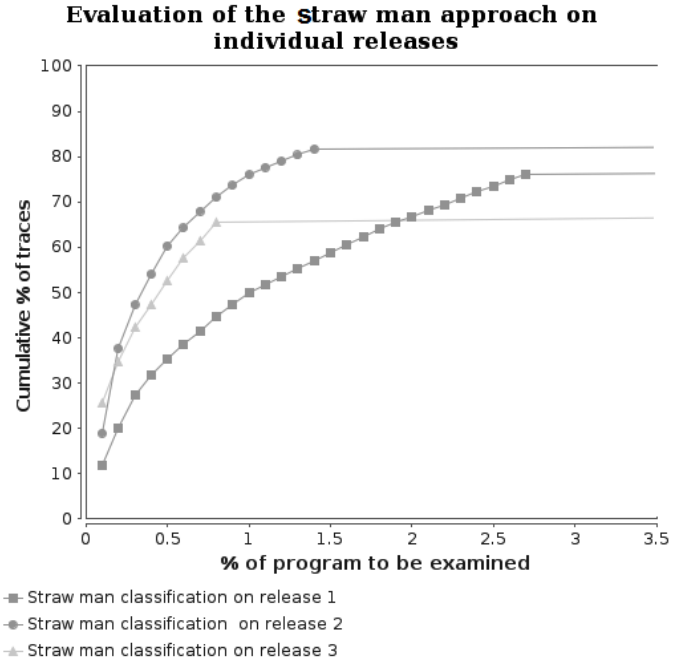

Fig. 4. Straw man approach on the three releases of the large commercial application by using 33% of the data for the training set, and 67% of the data for the test set. The horizontal axis is measured by using (2) with the total number of functions being 1,000.

D. Classifying Faulty Functions in Field Failure Traces across Releases

In this subsection, we show the results of our approach when we train F007 using traces from one release and attempt to diagnose recurring functions in traces from other releases. This ability would be useful in cases where software engineers want to diagnose recurring faulty functions from a release from which they do not have an established set of traces (e.g., a recently deployed release). In such a case, they can use previous releases to build the training models.

In Fig. 5, we show the results of the identification of recurring faulty functions in traces of release 2 by training F007 on release 1. There were about 15 common faulty functions in both release 1 and release 2. These 15 faulty functions were found faulty in 111 traces in release 2, and 33 traces in release 1. We trained F007 on 33 traces of release 1 to identify recurring faulty functions in 111 traces of release 2. Fig. 5 shows that faulty functions in 100 traces were discovered correctly out of 111 traces in release 2 (90% accuracy). This result required the review of less than 3% of the program functions. Similarly, in Fig. 5, we have also used the traces of both release 1 and release 2 to identify the faulty functions in the traces of release 3. There were about 15 common faulty functions in release 1, release 2, and release 3. The 15 faulty functions were found in 48 failure traces in

release 3, and 155 failure traces of release 1 and release 2. Fig. 5 shows that faulty functions in 47 traces in release 3 out of 48 traces (97% accuracy) were diagnosed by reviewing 4% of the program. In short, Fig. 5 shows that faulty functions across releases are identified accurately, especially when the number of traces is large (as in the case of release 3).

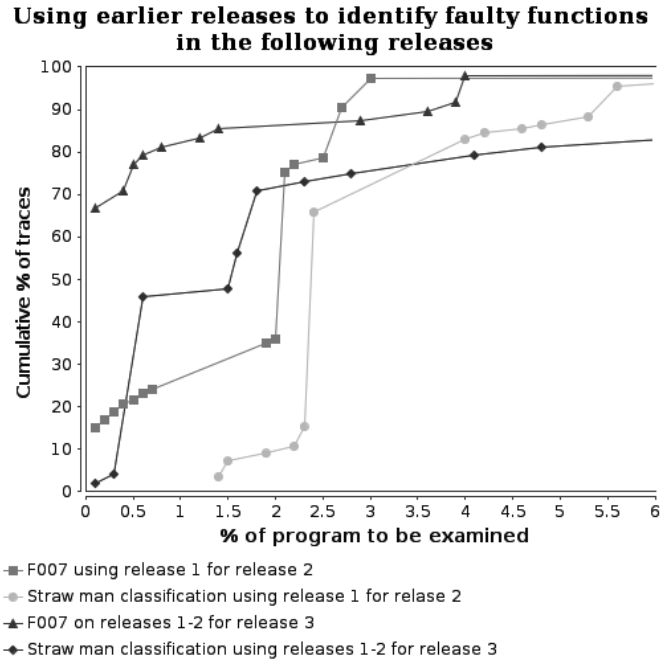

Fig. 5. Identifying the faulty functions across releases by using traces of earlier releases as the training set, and following releases as the test set.

Fig. 5 also provides a comparison of the straw man approach with F007. The results of the straw man approach were generated in the same way as mentioned earlier, except the traces of prior releases were used for ranking. Again the straw man approach requires much more code review, and faulty functions in fewer traces were identified. Fig. 5 also shows that, when there are more data for training, as in the case of release 1 and release 2 combined, then the accuracy of identifying faulty functions in failed traces with F007 is very high. For example, faulty functions in 67% of the failed traces were identified by reviewing only the first function (0.1% of the code) in the case of release 1 and release 2 in Fig. 5.

In reality, it is hard to know in advance whether a new trace is faulty due to known or unknown functions. There may exist traces of earlier releases, with some traces from the current releases. It is therefore important to also assess our approach in situations where we do not have a large set of traces from the current release. In Fig. 6, we show the results of training F007 on traces of earlier releases, and only 25% of traces from the current release. The 25% traces of the current release are sampled using the same stratification process mentioned in Section II.C. Fig. 6 shows that faulty functions in 72% to 82% of the system failure traces are again correctly diagnosed on the review of less than 1.6% of the program functions, which clearly demonstrates the effectiveness of our approach. We note that the accuracy of the identification of faulty functions is slightly lower in Fig. 6 compared to Fig. 3 when F007 was trained only on the current release. The reason is that F007 was trained on more traces (33%) from the current release in Fig. 3 compared to 25% of the traces from the current release in Fig. 6. We use different sets of traces to show different settings, and situations with few traces.

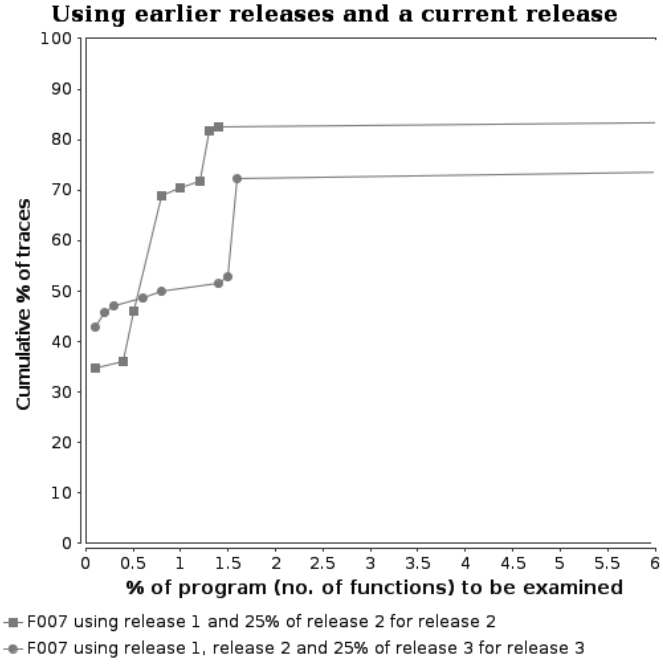

Fig. 6. Identifying faulty functions across releases using F007.

As a conclusion in this section, we showed that the functions that remain faulty across releases can be identified in new traces by using the traces of earlier releases of a software application with up to 97% accuracy. We also conclude that recurring faulty functions in field traces can be classified with approximately 90% accuracy by using system failure traces from all releases.

E. Comparing F007 to a clustering based approach

The closest techniques to our work are the clustering based approaches for field traces [3], [8], [10], [19], [33]. The majority of these techniques focus on clustering traces of crashing failures [3], [8], [10], [19]. They usually form clusters by measuring the similarity in function call sequences of top frames of stacks (functions that executed last). In our case, the traces were a combination of crashing and non-crashing failures, with the majority of them belonging to the non-crashing category. We mentioned in Section I that non-crashing failures are difficult to diagnose in function call traces. Faults causing crashing failures usually manifest themselves in functions that execute at the end of a function call trace. However, a fault causing a non-crashing failure can occur long before the manifestation of the fault as a failure. Dang et al. [8] report that, if a faulty function is in the middle of a trace, then their approach for clustering crashes results in misclassification. This problem makes clustering according to functions executed last not applicable for comparison with our

approach. Podgurski et al. [33] propose a technique of k-medoid clustering for non-crashing failures. Their intuition was that traces in each cluster would belong to the same faulty file. However, their clusters mostly represented multiple faulty files with some clusters containing up to eight files. Nonetheless, the work of Podgurski et al. [33] is the closest to our technique we have found.

For the sake of comparison with the clustering based approaches, we compared our approach against a clustering method. We applied k-medoid clustering to the traces by forming as many groups as there were classes (faulty functions) in the trace dataset. We employed Manhattan distance as the median based distance measure by using Weka [45]. The idea was that each group (cluster) would represent one class (i.e., faulty functions). We found that many clusters contained traces of more than one faulty function. To compare exactly with the F007 approach, a ranking method is required such that closely related clusters for a new trace in the test set can be predicted in an ordered list. No such ranking method exists in the literature. Therefore, we created a ranking approach based on clustering for the direct comparison of F007 against the clustering techniques.

We created a clustering based ranking on the basis of simple intuition. First, we clustered traces in the training set using k-medoid clustering with as many clusters (groups) as there were faulty functions. Second, we measured the Manhattan distance of a trace in the test set to all the formed clusters, and assigned the trace to a cluster with a minimum Manhattan distance. Third, we matched the faulty function of the test trace with one of the m faulty functions of the cluster that the trace was assigned to. If a match was found, then we considered that m functions were reviewed by a developer to discover the faulty function. Fourth, in the case of no match, we matched the faulty function of the trace with the faulty functions of other clusters one by one in decreasing order of the number of traces in the clusters. The intuition is that the developer would consider one of the faulty functions of the cluster with the largest number of traces as the suspected faulty function for the trace. When the match is not found, the developer would review faulty functions of the cluster with the second largest number of traces, and so on, to the last cluster. Fifth, the effort of the developer was measured by (2), the number of functions reviewed up to the diagnosis of the actual faulty function of the trace. In a similar manner to F007, we considered the total number of functions as 1,000 for (2). Fig. 7 shows the results of clustering based ranking on release 1 of the subject program. Fig. 7 also shows the results of F007 on the same release.

Observe from Fig. 7 that F007 can diagnose faulty functions in a trace with a smaller code review than clustering; only a few functions were required to be reviewed when F007 was used. We also observed similar results in the case of other releases of our large application with the clustering based ranking requiring review of more functions for the identification of the actual faulty function. It is possible that different clustering methods with different ranking heuristics can generate better or worst accuracy than the decision trees in F007. It is currently outside the scope of this paper to develop and test such intuitions as they don't exist in the literature. In future work, we expect to employ different clustering heuristics in F007 against the decision trees.

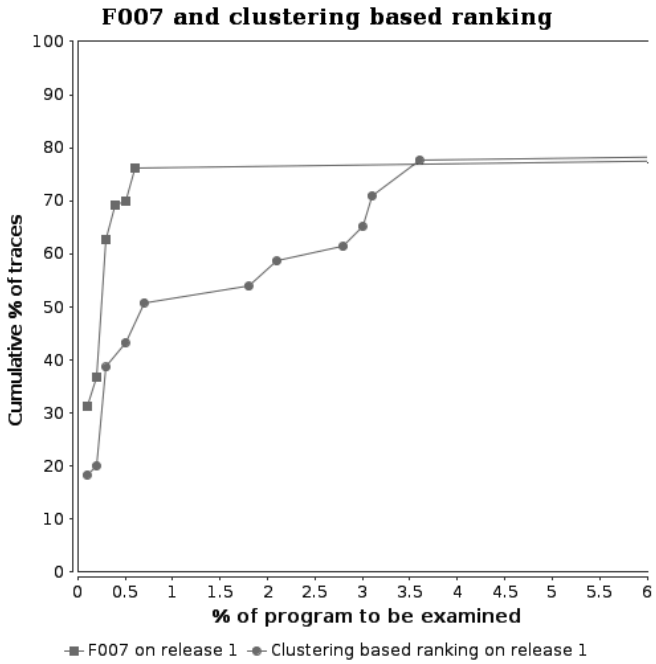

Fig. 7. F007 against an approach using ranking based on k-medoid clustering.

A limitation of F007 is that it does not diagnose the new faulty functions. The newer faulty functions can get masked as the older faulty functions. This limitation also exists in the clustering based approaches. A trace with a newer fault can be classified into already known clusters. Dang et al. report that traces with newer failures are put into new clusters based on the distance between new traces and traces in clusters. However, traces of newer crashes are still masked as the older crash type due to the similarity of function calls between the new traces and the traces in clusters. A workaround is to re-perform clustering or re-train F007 on the traces when there are a significant number of new traces. Another solution is to train a one-class classifier on all the failing traces, and use it to determine whether the fault is in a newer faulty function or older faulty function, and then apply F007 or clustering. The line without markers in Fig. 3 through Fig. 7 shows those traces which required reviewing all functions by a developer for fault diagnosis. This line without marker goes up to the last point (100,100) on the chart, but due to better visibility of the markers in the initial part of the chart we do not show the complete chart till the point (100,100). This line without markers actually shows those traces whose faulty functions did not exist in the training traces, and they were found faulty in the test traces. Thus, they required the engineer to review all the functions. Observe that these traces are few, and a majority of the traces have recurrent faulty functions.

F. Comparing F007 to a Manual Keyword Matching Approach

In industry, maintainers rely on historical databases to determine recurrent faults. When a new failure trace arrives, the maintainers search their historical database for symptoms common to the new failure trace. Maintainers search their

historical database with different keywords to find common symptoms. For example, the top (or first) function and the error codes in the new failure trace are the most commonly used keywords. In addition, maintainers may compare data patterns, rarely occurring functions with error codes, or any other symptoms that are unique to the new failure trace. If the symptoms match, the maintainers extract the fault identifier, and conclude that the fault is recurrent. The maintainers then extract fixes and necessary information using the fault identifier, and provide the information to the user reporting the failure. During this process, the maintainers may not have access to the source code, and it may be desirable to identify the recurrent fault without the source code. Also, it could be tedious to review the faulty function when the fault identifier and associated information are already available.

We therefore evaluated F007 by using fault identifiers as a label. We assigned each trace a label of fault identifier instead of a faulty function (see **Error! Reference source not found.**), and executed F007 on the dataset in a similar manner as described earlier. The results are shown in Fig. 8. The horizontal axis in Fig. 8 shows the percentage of fault identifiers required to be examined to diagnose the correct fault identifier. The Y-axis shows the cumulative percentage of failure traces used for testing. For example, the point (1, 52) shows that 52% of the failure traces were diagnosed correctly by reviewing 1% of the fault identifiers, which is the first fault identifier from F007's ranked list. The results of Fig. 8 are for release 1 of the subject program, and there were a total of 57 fault identifiers for this release.

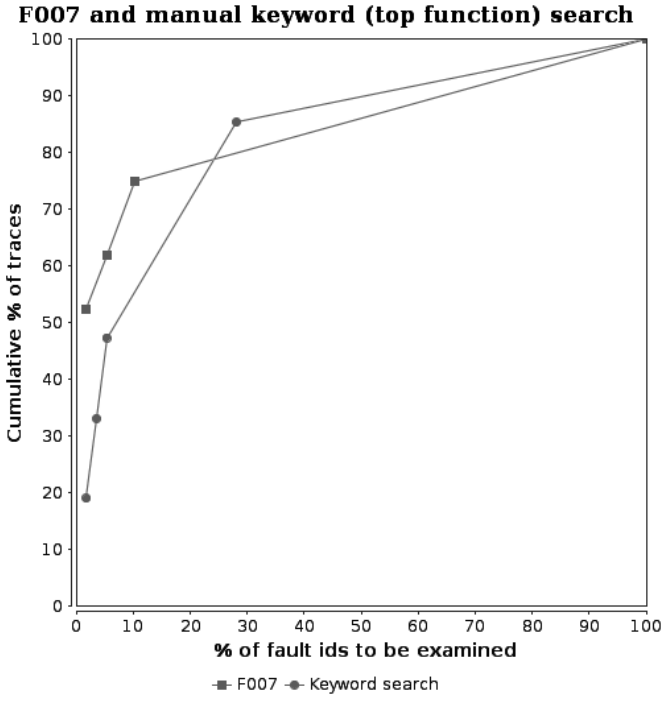

Fig. 8. F007 against the manual keyword based search.

In addition to F007, Fig. 8 also shows the results for a keyword based approach. In this case, we used the first function in the traces of the test set as the keyword to search the database of failure traces of the training set. We determined the fault identifiers when the keyword matched the first function of training traces. We then counted the number of fault identifiers and divided them by the total number of fault identifiers to determine the percentage of the fault identifiers reviewed to determine the correct fault identifier. If the keyword did not match in the training database, then we considered that the failure was not diagnosed.

It can be seen from Fig. 8 that F007 can diagnose fault identifiers with a higher accuracy than the manual keyword search approach. However, F007 diagnosed fewer failure traces than the keyword based approach. This result occurred because in some cases there was only one trace for training for a particular fault identifier, and F007 did not predict those fault identifiers. The keyword based approach was able to match the top function in those cases, and still was able to list the fault identifier for a search. However, the keyword based approach required reviewing a lot of fault identifiers to diagnose failures, whereas F007 allowed diagnosing the failure traces on the review of first few fault identifiers.

G. Discussion, and Lessons Learned

Our results on this large scale industrial software system show that different faults in a group of functions occur with similar function call traces. This result happens because we were able to identify the majority of recurring faulty functions with the same or a different fault by reviewing a small percentage of functions (e.g., 0.5% = 5 functions). The fact that few functions were required to be reviewed shows that traces of some faulty functions overlapped, but the traces were also distinct from the traces of some other faulty functions. If the traces had not overlapped, we would have had 100% accuracy on the review of the first suspected function. This result also shows that traces are not completely separable. In the case of multiple faulty functions, traces in the training set were labeled with multiple faulty functions, and classified in the test set with the label of multiple faulty functions. The traces of multiple faulty functions also overlapped with other traces, but at the same time they had enough distinguishability for identification. Therefore, only a few functions had to be considered for review before finding actual faulty functions.

During our experiments, we also observed that faulty functions persist across releases. This finding is consistent with another study in the literature [20]. Recurring faulty functions in the latest releases can be identified when a suitable number of traces of faulty functions in earlier releases are present.

Our experience with this large system shows that industrial traces can easily reach many Gigabytes, and are not trivial to parse. Data collection requires going through rigorous logistic checks, and the required data may not exist. Different sources have to be consulted to collect missing information. We have also found that sometimes the diagnosis of fault location in a failure trace can take several days to weeks. Often a developer may go through thousands of functions to find the root cause of a fault. F007 can be really helpful in such situations.

We have also found that many failure traces were related to configuration problems, such as the wrong network security protocol. The information about functions causing such faults was not kept in the database, and many traces were also purged. However, faulty components' names and fault identifiers were present for such faults. F007 could not be trained on faulty functions, but we evaluated F007 on fault identifiers and faulty components. F007 yields higher accuracy for fault identifiers and faulty components than for faulty functions because there are fewer fault identifiers or components than functions. Senior developers of this system suggest that the root cause of (configuration or in code) faults is mostly only a function. If F007 is trained with such information, then it can even facilitate users in solving recurring configuration problems themselves.

H. Limitations and Improvements

F007's effectiveness could be affected by large variations in the number of traces in the training set for different faulty functions. If the majority (e.g., 99%) of the training traces belongs to one or two faulty functions, and the remaining traces (e.g., less than 1%) belong to other faulty functions, then the decision tree would mostly predict the faulty functions associated with the majority (99%) of traces. This biased prediction is due to an imbalanced set of training traces. In machine learning, this condition is known as the imbalance class problem[45]. In such a case, a workaround would be to reduce the imbalance in traces of two faulty functions when training using the one-against-all approach. This workaround can be done by reducing the traces of the majority faulty functions in such a way that the percentage of traces of the minority faulty function increases to approximately 10%. This workaround can also be done by duplicating the number of traces of the minority faulty function such that they increase to approximately 10%.

F007's training could be constrained by the size of memory when there are hundreds of thousands of traces in the repository. In such cases, we must use parallel machine learning and computational techniques, such as Mahout[3] over Map Reduce [9], to train the decision tree algorithms on a large dataset of traces. In addition, we have also shown in Section III.B that the number of features can be significantly reduced without affecting the accuracy. This reduction also facilitates accommodating a large number of traces in memory during the training of the decision trees.

The software systems evolve over a period of time. Some of the functionalities that exist today may not be present in the future. An automated learning algorithm should be able to evolve over time with the software system. F007 can be improved by using an online or incremental learning algorithm. An incremental learning algorithm allows updating the learnt model on a new trace without having to use all the prior traces. The algorithm can also avoid additional memory overhead during training on many large size traces. Incremental learning is a separate research issue, and we consider it as future work for F007.

[3]https://mahout.apache.org/

IV. Threats to Validity

In this section, we describe certain threats to the validity of the research results. We classify threats into four groups: conclusion validity, internal validity, construct validity, and external validity [46].

A. Conclusion Validity

A threat to conclusion validity exists with traces of the number of faults we used to infer the conclusion. In the large software application, in Table III, we observed 82% recurring faults in the database, but we were able to collect traces of only some of the faults. The sample of system failure traces that we collected did not represent all the faults that occurred in the releases of the software application. In fact, the accuracy across releases would be higher if the failed traces of all the faults were used. This result happens because the decision tree would have had sufficient knowledge of faulty functions for which only one or two traces were present.

B. Internal Validity

A threat to internal validity exists in the implementation of this technique because it involved quite a lot of programming. We have mitigated this threat, and made our implementation reliable, by manually investigating the outputs.

C. Construct Validity

A threat exists in measuring the programmer's effort in discovering faulty functions. Recall, from Section II, that F007 generates a list of faulty functions for a new trace, and the programmer's effort is measured by counting the functions (or statements) examined. In a ranking based technique, such as F007, it is possible that two or more functions can be listed at the same rank. In such cases, the best case is the first function to be examined is faulty, and the worst case is the last function to be examined is faulty. This ordering implies that an incompetent technique will have a high best case accuracy (e.g., 90-100% accuracy on examining 1-10% of the program), and low worst case accuracy (e.g., 90-100% accuracy on examining 90-100% of the program), because it will list all the functions as faulty at the same rank. In our approach, the worst and the best case resulted in approximately the same accuracy. In a few results, there were measurable differences between the worst and the best case, but the difference was inconsequential. Thus, in all our results, we have shown the best case accuracies because the worst case was similar.

D. External Validity

We evaluated F007 on a commercial database application, mostly written in C and C++. F007 is still to be evaluated on other kinds of software applications before it can be generalized in all contexts.

V. Related Work

Scientific literature describes a number of fault discovery techniques. We classify the fault discovery techniques into two groups. Fault discovery techniques focusing on field failures, and fault discovery techniques focusing on in-house

failures. In the following sub-sections, we elaborate on each of these techniques.

A. Fault Discovery Techniques Focusing on Field Failures

Podgurski et al. [33] form clusters of execution traces of the field failures based on common faulty source files. The granularity in the Podgurski et al. approach is a faulty file, whereas a majority of the clusters contained failed traces with multiple files (fault origin), making it not suitable for the manual investigation of the correct faulty file (and the investigation of a finer-grain origin of a fault than just the faulty file gets even more difficult). In contrast, F007 discovers faults automatically at the finer-grained function-level; and the faults in the majority of traces can be discovered correctly by reviewing the first few suspected functions. Podgurski et al. experimented on GCC, Javac, and Jike; whereas, we experimented on a large industrial system of 20 million LOC. We have shown a comparison of F007 to a similar approach as Podgurski et al. in Section III.E.

Dang et al. [8] proposed a method to improve the grouping method of duplicate (recurrent) crash reports in a Windows error reporting system. Dang et al. measured the similarities of call stack traces by applying hierarchical clustering, and put the similar call traces in one group. Dhaliwal et al. [10] propose a two level grouping mechanism for Firefox based crash traces. The groups were formed on the basis of crash types by determining the similarities of top 10 functions in stacks using the Levenshtein distance. Brodie et al. [3] use string matching to group one function call trace of a crash with other groups of function-call traces for different crashes. The groups of crashes were formed by exactly matching the function call paths of different crashes. They claim that every group, formed on the basis of the same trace matches, has the same crashing reason. However, traces due to the same crashing reason (or the same fault) are not exactly the same, and they can take different approaches. Lee and Iyer [19] propose a technique to classify the recurrent crashing failures by literal matching of its function call trace with already known failure traces. They consider several heuristics to match several function call paths followed by the same fault. In F007, we model several paths leading to the same faulty function by the decision tree algorithm. These techniques focus on grouping function call traces of recurrent faults based on a similar crashing reason. F007 focuses on a finer grain identification of recurrent faulty functions from the system failure traces of crashes and non-crashes. F007 actually addresses a more difficult problem, like Podgurski et al. [33], of non-crashing failure classification. Non-crashing failures are more difficult to diagnose because a user may notice a failure well after the execution of the faulty code [33]. Examining the functions executed right before the system crashes may not be sufficient to diagnose a non-crashing fault.

Liu and Han [22] cluster failing runs according to a ranked list of assertions (i.e., check points) obtained using the statistical debugging tool SOBER [23]. They propose to collect passing and failing traces from the field to predict fault locations (assertions). They insert light weight assertions into code, collect traces, and if a fault is not found they insert new assertions. Collecting many passing and failing traces from the field can be detrimental to business operations due to the overhead of tracing. F007 focuses on a different problem of only recurrent faults from function call traces. These traces are commonly collected from the field. However, Liu and Han [22] use assertion based traces, which are not common. However, this technique [22] complements F007 by identifying new fault locations, and using F007 to instantly identify recurrent faults.

Xia et al. [52] combine genetic algorithm with multi-label classification algorithms to classify a failure trace into multiple fault types (i.e., multiple labels). In general, the basis of multi-label algorithms is to divide the multi-label dataset of traces into multiple datasets such that classification algorithms (e.g., SVM) can be trained in a normal manner on single labels. The predictions of individual classifiers are then combined using certain criterion by multi-label classification algorithms to get the final prediction. F007 also divides the dataset into multiple datasets but using the one-against-all approach. F007 treats multiple labels as one label during training (see Table I and Table II). F007 generates a ranking of the predictions, and evaluates the ranking by measuring the developers' effort; whereas Xin et al. [52] use the F-measure (which is similar to clustering techniques), and do not measure the developers' effort. It is unknown how much effort would be spent by developers in locating the correct label (faulty type) for a failure using the approach by Xin et al. In addition, Xin et al. [52] have been evaluated on smaller (approx. 1000 LOC) programs to medium size (approx. 10,000 LOC) programs; whereas, we have evaluated our approach on a large (approx. 20 million LOC) commercial program. The work by Xin et al. is related to our work, but it is not directly comparable.

Another statistical debugging tool, HOLMES [5], uses path profiles to classify faults for deployed software. Their technique can only be applied to the server side applications [5] because they have to redeploy software components with instrumentation of selected functions to collect the passing traces and the failing traces pertaining to one fault from the field. In some cases, this task may not be feasible for running servers as well due to the runtime redeployment of instrumented software components. F007, as mentioned earlier, focuses on a different problem of recurrent faults, and HOLMES can complement F007.

In [29], we proposed a technique to diagnose faulty functions from function call traces of crashing and non-crashing failures by using prior traces. We trained the C4.5 decision tree algorithm on function calls extracted from a collection of field traces to identify faulty functions in new traces. The study only focused on identifying recurrent faulty functions in the same release, and we evaluated it on small systems of 1000 LOC with traces collected using test suites. In this paper, we improve our previous work by empirically determining whether the same faulty functions across releases can be identified using prior traces. This improvement is important because faulty components persist across releases [20]. We also addressed the issue of the scalability of F007 by evaluating it on actual field traces of a very large industrial system of 20 million LOC, 200,000 functions, and traces of several Gigabytes (see Section III.B). We also discovered that the contribution of additional events (e.g., error codes, probe points, etc.) in function call traces of commercial software

applications other than function entry and exit events in diagnosis of faulty functions is not significant. In our earlier paper, the traces only contained function entry and exit events. In addition, we also showed that functions with smaller variations can be discarded without affecting the accuracy of fault diagnosis. This removal of error codes, probe points, function entry, and functions with small variations significantly reduces trace size and tracing overhead. It also facilitates efficient, scalable model generation (see Section III.B). Thus, significant improvements have been made from our earlier work [29].

Yuan et al. [53] employ support vector machines on system call traces to determine the root causes of configuration problems (e.g., network cable unplugged). Chen et al. [4] describe a technique based on the decision tree and association rule to diagnose configuration problems in large distributed systems (e.g., a faulty web server). Ding et al. [12] also propose a technique to identify faults occurring due to misconfiguration of a software system. In short, these techniques also complement our work in that they focus on the identification of faults in application interactions, or at a system level; whereas F007 focuses on the faults within an application.

Techniques for duplicate (recurrent) bug reports identification are also related to our work [31], [41]. Duplicate bug report identification techniques apply natural language processing techniques to comments from developers and users in bug reports to identify duplicate (recurrent) bug reports. F007 focuses on the identification of recurrent field traces, and the location of faults from field traces. In the automated collection of field traces, the reports may only contain function call executions without detailed information. Also, in contemporary bug report management systems (e.g., Firefox Socorro[4] and Bugzilla[5]), a bug report can be associated with many different crash traces, and a crash trace can be associated with many different bug reports. Therefore, related techniques focus separately on execution trace classification [8], [10], [33], [52], and bug report classification [31], [41]. F007 also focuses on execution trace classification, and does not assume that the detailed bug description is available.

Program comprehension techniques from software traces can be considered related [24]. Lo et al. [24] actually propose a close iterative pattern mining technique on software execution traces to facilitate developers' program comprehension. F007 however focuses on fault localization from software execution traces.

B. *Fault Localization Techniques focusing on In-house Failures*

Techniques for diagnosing fault locations by using the difference between passing traces and failing traces have also been a focus of many researchers. There are many examples of such techniques. Our first example is the execution slicing techniques that repeatedly compare pairs of passing and failing traces until the fault is found [1], [39], [47]. Our second example includes those techniques that present a ranked list of artifacts (e.g., statements, branches, etc.) to developers by contrasting passing and failing test case executions [17], [50], [40], [55], [6], [38]. Our third example is statistical debugging based techniques that present a ranked list of predicates (check points) by contrasting passing and failing runs of predicates [21], [23], [54], [56]. Our fourth example includes the use of neural networks [50] and chi-square based methods [48] on passing and failing statement level traces to generate a ranked list of faulty statements. Our fifth example includes those techniques that use passing and failing function call traces to generate a ranked list of faulty functions [11], and faulty classes for a fault [7]. Our sixth example includes the technique by Wang et al. [43] that combines existing fault localization measures (e.g., Tarantula [17] and Ochiai [38]) using genetic algorithm to improve the accuracy of fault localization. Our seventh example is about the technique by Lucia et al. [25] that investigates the 40 association measures, already found in the data mining literature, along with the measures by Tarantula and Ochiai on the localization of faults during software testing. Lucia et al. found that 50% of the association measures were able to find all the faults on the review of 25% to 27% of the program elements

To operate, these techniques require passing and failing traces. Collecting passing and failing traces from the field is not feasible due to tracing overhead, space requirements, and bandwidth consumption. In the case of multiple faults, these techniques usually require grouping traces due to the same faults (e.g., by clustering) to reduce a multiple faults problem to a set of single fault problems [49]. These techniques can be applied on field traces by reproducing the same faults on test machines. However, reproducing hundreds of failure reports with multiple faults can be time consuming. As we already know that 50% to 90% of field failures are due to previously known faults, then a wise approach to the fault localization of field failures is to use a technique like F007 to instantaneously identify recurring field faults, and use these techniques to identify new fault locations.

VI. Conclusions, and Future Work

Discovering the origin of a fault from field-failure reports is an arduous task, and can consume 30% to 40% of the time required to fix faults [35]. Despite knowing that 50% to 90% of field failures are due to the same faults [3], [19], [51], and 80% of the faults originate from 20% of the code [15], [32], the time and effort spent in identifying recurrent faults is still the same. Prior techniques in the literature focus on clustering stack traces of field crashes according to the similarity of faults [3], [19], [8], clustering crashing and non-crashing function call traces of field failures according to files [33], and using statistical debugging [5] on passing-failing traces to discover field faults (see Section V).

We propose a technique, F007, to identify recurring faulty functions in the traces of field failures. F007 trains the C4.5 decision tree algorithm on historical failure traces of current and prior releases. The trained decision trees are then used to identify faulty functions in new failure traces (see Section II). We evaluated F007 on a large industrial system of 20 million LOC and 200K functions (see Section III). Our results show that recurring faulty functions across releases and within the same release can be diagnosed with 90% accuracy on average

[4]https://bugzilla.mozilla.org
[5]https://crash-stats.mozilla.com/

(see Section III.C). Our results also show that different faults in related functions occur with similar function call traces, due to which recurring faulty functions can be easily identified (see Section III.E). The results also demonstrate that events like probe points, exceptions thrown, and functions with smaller variations do not contribute significantly to the automatic discovery of faulty functions. In fact, the size of traces can be reduced to approximately 50% by removing such events.

These results contribute to the body of knowledge by identifying recurring faulty functions in field failure traces across different releases. This approach is novel, and reduces the time spent in fault diagnosis during maintenance. This paper also contributes to the state of the art by demonstrating results on a large scale commercial application of 20 million LOC.

F007 has a limitation in that it cannot identify new faulty functions in field failure traces, and it only focuses on recurrent faulty functions. For future work, we are planning studies to overcome this limitation by integrating existing complementary techniques for in-house fault localization (see Section V.B).

REFERENCES

[1] H. Agrawal, J. R. Horgan, S. London, W. E. Wong, "Fault Localization using Execution Slices and Dataflow Tests," in *Proc. of International Software Symposium on Reliability Engineering*, France, Oct. 1995, pp.143-151.

[2] J.F. Bowring, J.M. Rehg, and M.J. Harrold, "Active Learning for Automatic Classification of Software Behavior," ACM *SIGSOFT Software Engineering Notes*, vol. 29, no. 4, pp. 195-204, Jul. 2004.

[3] M. Brodie, Sheng Ma, G. Lohman, L. Mignet, N. Modani, M. Wilding, J. Champlin, and P. Sohn, "Quickly Finding Known Software Problems via Automated Symptom Matching," in *Proc. of 2nd International Conference on Autonomic Computing*, Seattle, WA, June 2005, pp. 101-110.

[4] M. Chen, A. Accardi, E. Kiciman, A. Fox,, "Path-based Failure and Evolution Management," in *Proc. of International Symposium on Networked Systems Design and Implementation*, San Francisco, CA, March 2004, pp. 309-322.

[5] T. M. Chilimbi, B. Liblit, K. Mehra, A.V. Nori, and K. Vaswani, "HOLMES: Effective Statistical Debugging via Efficient Path Profiling," in *Proc. of 31st IEEE International Conference on Software Engineering*, Canada, May, 2009, pp. 34-44.

[6] H. Cleve and A. Zeller, "Locating Causes of Program Failures," in *Proc. of the 27th International Conference on Software Engineering*, St. Louis, MO, 2005, pp. 342-351.

[7] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight Defect Localization for Java," in *Proc. of 19th European Conference on Object-Oriented Programming*, Glasgow, UK, Aug. 2005, pp. 528-550.

[8] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel , "ReBucket: A Method for Clustering Duplicate Crash Reports Based on Call Stack Similarity," in *Proc. of the 34th International Conference on Software Engineering* , Zurich, Switzerland, 2012, pp. 1084-1093.

[9] J. Dean, and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of . ACM* , vol. 51, no. 1, pp. 107-113, Jan. 2008.

[10] T. Dhaliwal, F. Khomh, and Y. Zou, "Classifying Field Crash Reports for Fixing Bugs: A Case Study of Mozilla Firefox," in *Proc. of 27th IEEE International Conference on Software Maintenance* (ICSM), 2011, pp. 333-342.

[11] G. Di Fatta, S. Leue, and E. Stegantova, "Discriminative Pattern Mining in Software Fault Detection." in *Proc. of 3rd International Workshop on Software Quality Assurance*, Oregon,, Nov. 2006, pp. 62-69.

[12] X. Ding, H. Huang, Y. Ruan, A. Shaikh, and X. Zhang, "Automatic Software Fault Diagnosis by Exploiting Application Signatures," in *Proc. of 22nd Conference on Large Installation System Admini*stration, San Diego, CA, Nov. 2008, pp. 23-39.

[13] A. Hamou-Lhadj, "Techniques to Simplify the Analysis of Execution Traces for Program Comprehension," Ph.D. Dissertation, School of Information Technology and Engineering (SITE), University of Ottawa, 2003

[14] A. Hamou-Lhadj, and T. C. Lethbridge, "Techniques for Reducing the Complexity of Object-Oriented Execution Traces," in *Proc. of the 1th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, Amsterdam, Netherlands, 2003, pp. 35-40.

[15] M. Gittens, Y. Kim, and D. Godwin, "The Vital Few Versus the Trivial Many: Examining the Pareto Principle for Software," in *Proc. of 29th International Conference on Computer Software and Applications*, Edinburgh, Scotland, July 2005, pp. 179-185.

[16] D. Hare, and D. Julin. (2007, April). "The Support Authority: Interpreting a WebSphere Application Server trace file," *IBM WebSphere Developer Technical Journal*. [Online]. Available:http://www.ibm.com/developerworks/websphere/techjournal/0704_supauth/0704_supauth.html

[17] J. A Jones, and M. J. Harrold, "Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique," in *Proc. of 20th International Conference on Automated Software Engineering.*, CA, USA, 2005, pp.273-282.

[18] S. Kulkarni, "Software Defect Rediscoveries: Causes, Taxonomy and Significance," MS thesis, Department of Computer Science, The University of Western Ontario, 2008.

[19] I. Lee, and R. Iyer, "Diagnosing Rediscovered Problems Using Symptoms," *IEEE Transactions on Software Engineering*., vol. 26, no. 2, pp.113-127, Feb. 2000.

[20] Z. Li, N. H. Madhavji, S.S. Murtaza, M. Gittens ., "Characteristics of Multiple-Component Defects and Architectural Hotspots: A Large System Case Study,"

Empirical Software Engineering (ESE), vol. 16, no. 5, pp. 667-702, Oct. 2011.

[21] B. Liblit, A. Aiken, A.X. Zheng, and M. I. Jordan, "Bug Isolation via Remote Program Sampling," in *Proc.of ACM SIGPLAN 2003 Conference on Prog. Language Design and Implementation (PLDI '03)*, May 2003, pp. 141-154.

[22] C. Liu, and J. Han, "Failure Proximity: A Fault Localization-based Approach," in *Proc. of the 14th SIGSOFT Symposium on Foundations of Software Engineering*, Portland, OR, Nov. 2006, pp. 45-56.

[23] C. Liu, X. Yan, L. Fei, J. Han, S. P. Midkiff, "SOBER: Statistical Model-Based Bug Localization," *SIGSOFT Software Engineering Notes*, vol. 30, no.5, , pp. 286-295, Sep. 2005.

[24] D. Lo, S. Khoo, and C. Liu, "Efficient Mining of Iterative Patterns for Software Specification Discovery." in *Proc. of the 13th International Conference on Knowledge Discovery and Data Mining (KDD)*. 2007, pp. 460-469.

[25] Lucia, D. Lo, A. Jiang, F. Thung, A. Budi, "Extended Comprehensive Study of Association Measures for Fault Localization." *Journal of Software: Evolution and Process*, vol. 26, no. 2, pp. 172-219, Feb. 2014.

[26] R. B. Melnyk. (2004, Sep.). "DB2 Basics: An introduction to the DB2 UDB trace facility," *DB2 Information Development*, IBM Canada Ltd. [Online]. Available: http://www.ibm.com/developerworks/data/library/techarticle/dm-0409melnyk/index.html

[27] A.,V. Miranskyy, M. Davison, M. Reesor, and S. S. Murtaza, "Using Entropy Measures for Comparison of Software Traces," *Journal of Information Sciences*, vol. 203, pp. 59-72, Oct. 2012.

[28] Mozilla Crash Statistics, [Online]. Available : http://crash-stats.mozilla.com

[29] S.S. Murtaza, M. Gittens, Z. Li, and N. H. Madhavji, "F007: Finding Rediscovered Faults from the Field using Function-level Failed Traces of Software in the Field," in *Proc. of Conf. of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, Toronto, Canada, Nov. 2010, pp. 61-75.

[30] S.S. Murtaza, A. Rehman, A. Hamou-Lhadj, and M. Couture, "On the Comparison of User-space and Kernel-space Traces in Identification of Software Anomalies," in *Proc. of 16th Conference on Software Maintenance and Reengineering*, Hungary, *2012*, pp. 127-136.

[31] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun, "Duplicate Bug Report Detection with a Combination of Information Retrieval and Topic Modeling," in *Proc. of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2012. pp. 70-79.

[32] T. J. Ostrand, E. Weyuker, and R.M. Bell, "Predicting the Location and Number of Faults in Large Software Systems," *IEEE Transactions on Software Engineering*, vol. 31, no. 4, pp. 340-355, April 2005.

[33] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, and J. Sun, "Automated Support for Classifying Software Failure Reports," in *Proc. of International Conference on Software Engineering (ICSE).*, Portland, OR, 2003, pp. 465-475.

[34] K. Polat, and S. Güneş, "A Novel Hybrid Intelligent Method Based on C4.5 Decision Tree Classifier and One-Against-All Approach for Multi-class Classification Problems," *Journal of Expert Systems with Applications.*, vol. 36, no.2, Pergamon Press, pp.1587-1592, Mar. 2009.

[35] Proprietary Workshop on Large Commercial Software, London, Canada: The University of Western Ontario, Sep. 2008.

[36] F. Provost, and P. Domingos, "Tree Induction for Probability-based Ranking," *Machine Learning*, vol. 52, no. 3, pp. 199-215, Sep. 2003.

[37] J. R. Quinlan, *C4.5: Programs for Machine Learning*, San Francisco, CA: Morgan Kaufmann Publishers, 1993.

[38] R. Abreu, P. Zoeteweij, R. Golsteijn, and A.V. Gemund, , "A Practical Evaluation of Spectrum-based Fault Localization," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1780-1792, Nov. 2009.

[39] M. Renieres, and S.P. Reiss, "Fault Localization with Nearest Neighbor Queries," in *Proc. of the 18th International Conference on Automated Software Engineering*, Montreal, Canada, 2003, pp. 30-39.

[40] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold , "Lightweight Fault-localization Using Multiple Coverage Types," in *Proc. of the 31st International Conference on Software Engineering.*, Vancouver, Canada, 2009, pp. 56-66.

[41] Y. Tian, C. Sun, and D. Lo, "Improved Duplicate Bug Report Identification," in *Proc. of 16th European Conference on Software Maintenance and Reengineering (CSMR)*, March 2012, pp.385-390.

[42] *Ubuntu Apport Crash Reporting*, [Online]. Available: https://wiki.ubuntu.com/Apport

[43] S. Wang, D. Lo, L. Jiang, Lucia, H.C. Lau, "Search-based Fault Localization," in *Proc. of 26th IEEE/ACM International Conference on Automated Software Engineering*, 2011, pp.556-559.

[44] WER, Windows Error Reporting, [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/hardware/gg487440.aspx

[45] I.H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques*, San Francisco, CA: Morgan Kaufmann, 2005.

[46] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén , *Experimentation in Software Engineering: An Introduction*, Norwell, MA: Kluwer Academic Publishers, 2000.

[47] W. E. Wong, and Y. Qi, "Effective Program Debugging Based on Execution Slices and Inter-Block Data Dependency," *Journal of System and Software*, vol.79, no. 7, pp. 891-903, July 2006.

[48] W.E. Wong, T. Wei, Y. Qi, and L. Zhao, "A Crosstab-based Statistical Method for Effective Fault Localization," in *Proc. of 1st International Conference*

on Software Testing, Verification and Validation, Norway,2008, pp. 42-51.

[49] W.E. Wong, V. Debroy, R. Golden, Xu. Xiaofeng, B. Thuraisingham, "Effective Software Fault Localization Using an RBF Neural Network," *IEEE Transactions on Reliability*, vol. 61, no. 1, pp. 149-169, March 2012.

[50] W.E. Wong, Y. Qi, L. Zhao, Kai-Yuan Cai , "Effective Fault Localization using Code Coverage," in *Proc. 31st IEEE Int'l Conf. on Computer Software. & Application.*, China, July 2007, pp.449-456.

[51] A. Wood, "Software Reliability from the Customer View," *Computer*, vol. 36, no. 8, pp.37-42, Aug. 2003.

[52] X. Xia, Y. Feng, D. Lo, Z. Chen, and X. Wang , "Towards more accurate Multi-label Software Behavior Learning," in *Proc. of IEEE International Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*, Feb. 2014, pp.134,143.

[53] C. Yuan, N. Lao, J. Wen, J. Li, Z. hang, Y. Wang, and W. Ma, "Automated Known Problem Diagnosis with Event Traces," *SIGOPS, OS. Syst. Rev.*, vol. 40, no. 4, pp. 375-388, Aug. 2006.

[54] Z. Zhang, W. K. Chan, T. H. Tse, Y. T. Yu, and P. Hu, "Non-parametric Statistical Fault Localization". *Journal of Systems and Software*, vol. 84, no. 6 , pp. 885-905, June 2011.

[55] Z. Zhang, B. Jiang, and X. Wang, "Capturing Propagation of Infected Program States," in *Proc. of International Conference on Foundations of Software Engineering.*, Netherlands, 2009, pp. 43-52.

[56] A.X. Zheng, M.I. Jordan, B. Liblit, and A. Aiken, "Statistical Debugging of Sampled Programs," *Advances in Neural Information Processing Systems*, , pp. 9-18, 2004.

ACKNOWLEDGMENT

We are thankful to Mark Wilding, Andriy Miranskyy and Dave Godwin of IBM for their technical support and intuitive insights during this study.

Syed Shariyar Murtaza received his Ph.D. from the University of Western Ontario in 2011. He received his MS in Computer Engineering from Kyung Hee University in 2006 and BS from the University of Karachi in 2004. He has been working as a researcher and a software engineer with Concordia University and Defence Research and Development Canada since 2011. He specializes in the applications of machine learning in software engineering, data analytics, and information management

Nazim H. Madhavji is a Professor in the Department of Computer Science at the University of Western Ontario, Canada. He is particularly known for his contributions to the knowledge on interactions between system requirements and architectures, and for his work on the impediments to regulatory compliance in large projects, execution trace analysis, defect analysis, the evolution of systems, software quality, the congruence between software products and processes, and empirical studies.

Mechelle Gittens has worked and carried out research in software engineering since 1995. She is currently a Lecturer at the University of the West Indies – Cave Hill Campus where she teaches and does research in Computer Science. Mechelle has a Master's and Doctorate from the University of Western Ontario (UWO), where she is now a Research Adjunct Professor. Her work is in software quality, quality of life technologies, software testing, empirical software engineering, software reliability, and project management. She has published at several international forums in these areas, and jointly holds a US patent in software testing.

Abdelwahab Hamou-Lhadj is a tenured associate professor in ECE, Concordia University. His research interests include software modeling, software behavior analysis, software maintenance and evolution, anomaly detection systems. He holds a Ph.D. degree in Computer Science from the University of Ottawa (2005). He is a Licensed Professional Engineer in Quebec, and a long-lasting member of IEEE and ACM.