Scaffle: Bug Localization on Millions of Files

Michael Pradel*
University of Stuttgart
Germany

Vijayaraghavan Murali
Facebook
USA

Rebecca Qian
Facebook
USA

Mateusz Machalica
Facebook
USA

Erik Meijer
Facebook
USA

Satish Chandra
Facebook
USA

ABSTRACT

Despite all efforts to avoid bugs, software sometimes crashes in the field, leaving crash traces as the only information to localize the problem. Prior approaches on localizing where to fix the root cause of a crash do not scale well to ultra-large scale, heterogeneous code bases that contain millions of code files written in multiple programming languages. This paper presents Scaffle, the first scalable bug localization technique, which is based on the key insight to divide the problem into two easier sub-problems. First, a trained machine learning model predicts which lines of a raw crash trace are most informative for localizing the bug. Then, these lines are fed to an information retrieval-based search engine to retrieve file paths in the code base, predicting which file to change to address the crash. The approach does not make any assumptions about the format of a crash trace or the language that produces it. We evaluate Scaffle with tens of thousands of crash traces produced by a large-scale industrial code base at Facebook that contains millions of possible bug locations and that powers tools used by billions of people. The results show that the approach correctly predicts the file to fix for 40% to 60% (50% to 70%) of all crash traces within the top-1 (top-5) predictions. Moreover, Scaffle improves over several baseline approaches, including an existing classification-based approach, a scalable variant of existing information retrieval-based approaches, and a set of hand-tuned, industrially deployed heuristics.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; *Software creation and management*;

KEYWORDS

Bug localization, software crashes, machine learning

ACM Reference Format:
Michael Pradel, Vijayaraghavan Murali, Rebecca Qian, Mateusz Machalica, Erik Meijer, and Satish Chandra. 2020. Scaffle: Bug Localization on Millions of Files. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3395363.3397356

*Work mostly performed while on sabbatical at Facebook, Menlo Park.

ISSTA '20, July 18–22, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8008-9/20/07…$15.00
https://doi.org/10.1145/3395363.3397356

1 INTRODUCTION

When software crashes in the field, often the only information available to developers are *crash traces*. Such traces come in various forms and may include various kinds of hints about the code that causes the crash, including stack traces, error messages, information about the program state just before the crash, and additional information added by tools that process crash traces. To prevent future instances of the same crash, developers must identify where exactly in the code base to fix the underlying problem – a problem referred to as *bug localization*.

1.1 Crash-Based Bug Localization at Scale

We focus on localizing bugs at the file-level within ultra-large scale, heterogeneous code bases, i.e., code bases that contain multiple millions of code files written in multiple programming languages. Addressing the file-level bug localization problem at scale can help with the process of handling field crashes in multiple ways. By pinpointing which files are most relevant for a crash, bug localization helps find the right team or person to address the crash. Finding the right developer for a given crash is non-trivial in a large organization, and associating crashes to the wrong developers consumes valuable time and resources. Once the right developer has been identified, knowing the buggy file helps the developer to focus on where to implement a fix of a crash. Moreover, identifying the file to fix can serve as a first step in automated program repair [14].

Unfortunately, manual bug localization is difficult and time-consuming. One reason is that crash traces contain lots of noise not directly related to the bug location. Another reason is that, for very large-scale code bases, there may be millions of code files to choose from. The problem is further compounded by the fact that widely deployed software may lead to such a volume of crashes per day that is practically impossible to process without appropriate tools.

1.2 Scalability Problems of Prior Work

Prior work has proposed several automatic bug localization techniques, which are extremely inspiring, but not easily applicable to ultra-large scale, heterogeneous code bases, such as those that motivate this work. Broadly speaking, one group of prior work is based on traces of correct and buggy executions [1, 6, 15]. These approaches assume that a test suite is available where some tests pass while other tests fail due to the bug. Unfortunately, failing tests

Table 1: Prior work on bug localization.

Work	Evidence of bug	Technique	Lang.	Code analysis	Scalability issue
[18]	Bug report	Topic modeling	Java	Yes	Extracts topics from each source file.
[33]	Stack trace	IR	Java	Yes	Extracts natural language words from each source file.
[21]	Bug report	IR	Java	Yes	Parses and analyzes each source file.
[25]	Set of stack traces	Clustering & learning to rank	Java, C/C++	No	Queries a learned model for each pair of crash and source file.
[9]	Bug report	Classifier	C/C++	No	Classifier where each source file is a separate class.
[30]	Set of stack traces	Static analysis & heuristics	C/C++	Yes	Call graph analysis, control flow analysis, and slicing for each source file.
[32]	Bug report	Learning to rank	Java	Yes	Extracts natural language words from each source file.
[27]	Bug report	IR	Java	Yes	Analyzes and segments each source file.
[17]	Bug report and stack trace	Static analysis & IR	Java	Yes	Analysis of control flow and data flow dependencies.
[13]	Bug report	Learning & IR	Java	Yes	Extracts features from each source file.
[19]	Bug report	IR	Java	Yes	Extracts terms from each source file.
[12]	Bug report	Multiple classifiers	Java	Yes	Analyzes all source files; queries learned models for each pair of bug and source file.
This work	Raw crash traces	Learning & IR	Lang.-indep.	No	None

that reproduce field crashes often are not available, and generating such tests is a challenging problem on its own [22].

The other group of prior work is based on some evidence of the bug, such as a stack trace produced by the bug or a natural language bug report. Table 1 summarizes the most closely related approaches in this group. Many existing approaches statically analyze each file in the code base, e.g., to analyze dependencies between code elements or to extract features of individual files [12, 13, 17–19, 21, 27, 30, 32, 33]. The extracted information can then, e.g., be fed into an information retrieval (IR) component that compares the words in a bug report or stack trace with the tokens in code files [13, 17, 19, 21, 27, 33]. Another direction addresses bug localization as a learning problem, e.g., via a classification model that considers each file in the code base as a separate class [9].

The main limitation of these previous approaches is the *lack of scalability* to ultra-large scale, heterogeneous code bases. Analyzing each source file in a code base that contains millions of files written in multiple languages is far from trivial, even with a lightweight analysis. None of such approaches [12, 13, 17–19, 21, 27, 30, 32, 33] has been applied at the scale targeted here. To the best of our knowledge, the largest code base used in prior work [12] consists of 71,000 Java files from 45 projects, i.e., two orders of magnitude smaller than the multi-million code base considered here. The approaches that do not require any source file analysis suffer from their own scalability issues. Work that considers each source file as a separate class for a classifier [9] does not scale well, as we show experimentally in Section 6.5. Other work, which queries a trained model for each pair of a crash and a source file [25], takes at least a linear (w.r.t. the number of source files) amount of time for each individual crash.

Moreover, almost all existing techniques (except [25]) target stack traces produced by a single programming language, typically Java, building on parsers, regular expressions, and heuristics specialized for stack traces in this language. Unfortunately, a single-language approach is difficult to adopt to crash traces that originate from several programming languages, come in various different formats, and may have been processed by a diverse and evolving set of tools.

1.3 Our Work in a Nutshell

This paper presents Scaffle, the first bug localization technique for crashes caused by code in ultra-large scale, heterogeneous code bases. To scale Scaffle to code bases with multi-million files, and tens of thousands of crashes, we use the key insight that *the problem of localizing bugs based on crash traces can be decomposed into two sub-problems*. Figure 1 shows a high-level overview of this decomposition. Existing bug localization techniques (top) address the problem through an end-to-end approach that directly compares the crash trace to files in the code base. Instead, Scaffle (bottom) decomposes the problem into two sub-problems, each addressed by a separate component.

The first component of Scaffle addresses the problem of identifying the most relevant lines of a given crash trace. This component, called the *trace-line model*, is a machine learning model that reads all lines of the given trace and assigns a relevance score to each line. We implement the trace-line model using neural network-based supervised learning, which learns from past crashes and the bug fix locations associated with them. The second component of Scaffle addresses the problem of matching the most relevant lines in a crash trace with file paths in the code base. We view this problem as an IR problem where a line of a crash trace serves as a query over the file paths in the code base. Decomposing the problem is inspired by the observation that a single line in a crash trace often provides most hints to localize the buggy file, while the other lines add various kinds of noise.

Our work breaks with two assumptions made by prior work. First, the approach does not assume that one can analyze the entire code base. Instead, Scaffle matches specific lines in a crash trace with paths in the code base, without ever analyzing the content of the files stored at these paths. In particular, Scaffle avoids statically analyzing all files in the code base. Second, the approach does not assume a specific programming language, and as a result, also does not assume a specific structure or format of crash traces. Instead of building, fine-tuning, and constantly evolving specialized parsers for different trace formats, the model automatically learns to parse and understand a diverse set of crash traces. As a result of these design decisions, the approach scales well to very large code bases that contain code written in multiple languages.

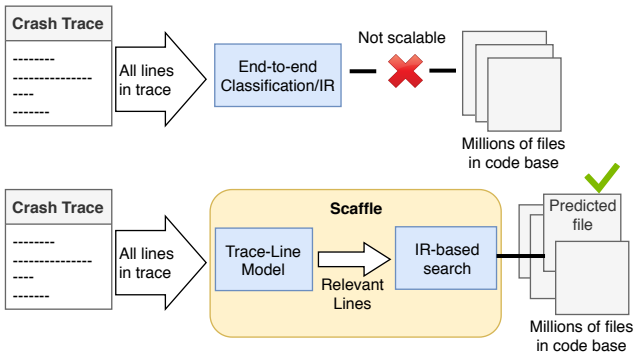


Figure 1: Overview of Scaffle (bottom) and end-to-end methods (top).

We evaluate our work by applying it to tens of thousands of crash traces produced by a large-scale code base at Facebook, which contains several millions of files. To the best of our knowledge, our dataset is an order of magnitude larger than previous evaluations of crash trace-based bug localization techniques, and we apply Scaffle to a code base that is at least two orders of magnitude larger than in any prior work [12].

The empirical results show that decomposing the bug localization problem into two sub-problems is key to obtaining an approach that is effective for a large-scale code base. We find that the model is effective at predicting those lines in a trace that pinpoint the buggy file, reaching a mean reciprocal rank of over 0.8. Using this model, Scaffle correctly predicts a to-be-fixed file for 40% to 60% (50% to 70%) of all crash traces in the top-1 (top-5) predictions. The effectiveness is roughly the same across different programming languages and parts of the code base.

Comparing Scaffle to prior work highlights that end-to-end classification [9] does not scale well to millions of files, and that a baseline (relying on no source analysis) end-to-end IR-based bug localization in the vein of prior work [13, 17, 19, 21, 27, 33] is less effective than our approach.

In summary, this paper makes the following contributions:

- A scalable, language-independent technique to predict from raw crash traces which files to change to address a crash.
- The insight that the crash-based bug localization problem can be decomposed into two simpler problems: identifying relevant lines in a crash trace and matching those lines with file paths in the code base.
- A learned model for predicting the most relevant lines of crash traces. The model is realized as a neural network that is trained in a supervised manner on past fix locations.
- Empirical evidence from applying our approach in a large-scale, industrial setting, showing that Scaffle effectively predicts those parts of the code base to focus on to fix a crash.

2 OVERVIEW AND EXAMPLE

Before describing the details of Scaffle in Sections 3 and 4, this section illustrates the main ideas with an example. The inputs given to Scaffle are a *raw crash trace* and the set of all file paths in the code base. By raw crash trace, we mean any kind of structured or unstructured text that we assume to be separated into lines. In particular, these traces may contain one or more stack traces, information about the application and the underlying system where the crash occurred, and information about the state of the application. Figures 2(a) and (b) give two examples of crash traces, one produced by a crash in Java code and one produced by a crash in PHP code. We assume a single representative crash trace as the input to Scaffle, e.g., identified by techniques for clustering and prioritizing crashes [3–5, 8, 10, 23]. The file paths in the code base each are a sequence of path segments, e.g., "proj/pkg/someFile.java".

The first component of Scaffle, the trace-line model, identifies those lines of a crash trace that are most relevant for localizing the bug. To obtain this component, we exploit the insight that most large-scale projects have plenty of historical data about crashes and about code changes that fix the root cause of crashes. Given the fix location of a crash, we derive how relevant each line of the crash trace is for finding the bug location, based on whether the line contains parts of the file path of the bug location. Scaffle then learns from this data a machine learning model that summarizes individual lines of a crash trace and then predicts the relevance of each of these lines (Section 3). Learning a model, instead of hard-coding a set of heuristics, addresses the challenge that the format of crash traces not only varies, but also evolves over time, as new programming languages and APIs become popular.

For the examples in Figure 2(a) and (b), the right-most column shows the relevance score that our neural trace-line model predicts for each line. Intuitively, the lines with highest relevance contain the most information about file paths likely to be changed to fix the bug. As illustrated by Figure 2(a), some of the most relevant lines may be among the first lines of a crash trace, e.g., because it summarizes the location where an exception was thrown. As illustrated by Figure 2(b), the most relevant lines may also be somewhere in the middle of a crash trace. Section 3 describes in detail the trace-line model that predicts which of the given lines are most relevant.

The second component of Scaffle is an IR-based search for the most likely bug locations. In IR, a query usually needs to be matched against a possibly large set of documents. In our case, one of the most relevant lines of the crash trace is the query, and each file path in the code base is a document. Specifically, we tokenize all file paths into segments, and similarly tokenize the words appearing in the most relevant line. We then feed it as the query into an IR-based search engine, and return the file paths from the search results as a ranked list of possibly buggy files. The underlying assumption is that the most relevant trace line often contains some path segments of the buggy file, even though it may not refer to the exact path.

For our running example, line 28 of the Java trace in Figure 2(a), i.e., the most relevant line, will be tokenized into the keywords "at dostuff dostuffbrowsercontroller exitdomorestuff dostuffbrowsercontroller java" for the query. For this query, a path "projectX/packageA/dostuff/DoStuffBrowserController.java" would be considered more similar to the query than a path "projectX/packageB/some/other/path/Logger.java", and hence will be predicted as the most likely path. Section 4 describes our approach for matching trace lines with file paths in detail.

(a) Example of a Java crash trace.

Nb.	Lines of raw crash trace	Predicted relevance
1	`android_crash:java.lang.IllegalStateException:dostuff.DoStuffBrowserController.clearBrowserFragment`	83%
2	`stack_trace: java.lang.RuntimeException: Unable to destroy activity {lala/lala.LoginActivity}: ...`	58%
3	`at android.app.ActivityThread.performDestroyActivity(ActivityThread.java:3975)`	14%
...	(dozens of more lines)	
23	`Caused by: java.lang.IllegalStateException: Activity has been destroyed`	12%
24	`at android.app.FragmentManagerImpl.enqueueAction(FragmentManager.java:1376)`	21%
25	`at android.app.BackStackRecord.commitInternal(BackStackRecord.java:745)`	12%
26	`at android.app.BackStackRecord.commitAllowingStateLoss(BackStackRecord.java:725)`	30%
27	`at dostuff.DoStuffBrowserController.clearBrowserFragment(DoStuffBrowserController.java:775)`	85%
28	`at dostuff.DoStuffBrowserController.exitDoMoreStuff(DoStuffBrowserController.java:196)`	100%
29	`at domorestuff.DoMoreStuffRootView.exitDoMoreStuff(DoMoreStuffRootView.java:554)`	87%
...	(dozens of more lines)	
76	`app_upgrade_time: 2018-08-19T17:02:12.000+08:00`	5%
77	`package_name: lala`	1%
78	`peak_memory_heap_allocation: 92094532`	1%
79	`app_backgrounded: false`	13%
...	(dozens of more lines)	

(b) Example of a PHP crash trace.

Nb.	Lines of raw crash trace	Predicted relevance
1	`[twi01447.08.ftw1.example.com] [Sat Sep 29 13:20:40 2018] ...`	9%
2	`(Events: <null_response_query_id>)`	2%
3	`(App Version: 189.0.0.44.93)`	0%
4	`(NNTraceID: FAiHFWy4Isj)`	1%
5	`(Sampling ID: A_oSj-oZbo1GJnM8JHKL3o_)`	1%
...	(dozens of other lines)	
31	`trace starts at [/var/abc/def/core/runtime/error.php:1021]`	35%
32	`#0 __log_helper(...) called at [/var/abc/def/core/runtime/error.php:1021]`	47%
33	`#1 log() called at [/var/abc/def/core/logger/logger.php:1162]`	49%
34	`#2 NNDefaultLogMessage->log() called at [/var/abc/def/core/logger/logger.php:938]`	41%
35	`#3 NNLogMessage->process() called at [/var/abc/def/core/logger/logger.php:804]`	46%
36	`#4 NNLogMessage->warn() called at [/var/abc/def/core/logger/logger.php:791]`	65%
37	`#5 NNLogMessage->warn_High() called at [/var/abc/def/foo/multifoo/client/base/MultifooClient.php:1518]`	82%
38	`#6 MultifooClient->genPopulateResults$memoize_impl() called at [/var/abc/def/foo/multifoo/client/base/MultifooClient.php:2248]`	82%
39	`#7 MultifooClient->genQueryID() called at [/var/abc/def/bar/query/multifoo/MultifooQuery.php:5782]`	77%
40	`#8 Closure$MultifooQuery::genStuff#9() called at [/var/abc/def/gates/core/Gate.php:390]`	68%
...	(dozens of other lines)	

Figure 2: Examples of Scaffle's approach to crash-based bug localization (Note: The traces and file paths are made-up but modeled after real data.)

3 PREDICTING RELEVANT LINES IN RAW TRACES

This section describes Scaffle's approach for predicting the most relevant lines in a given crash trace, which is the first of two steps for predicting the bug location. Our approach is based on the observation that for many crash traces, a single line is sufficient to pinpoint the bug location, while dozens or even hundreds of other lines are irrelevant. The reason may be, e.g., that the bug location is on the stack when the crashes happens, and hence occurs in a single frame of the stack trace, or that the crash trace mentions a specific term that matches the buggy path.

Scaffle trains a machine learning model to identify the most relevant lines in a trace. The motivation for choosing a learning-based approach over, e.g., hard-coding heuristics for a specific trace format, is two-fold. First, learning from data allows the approach to cover traces produced by multiple programming languages and coming in different formats. Second, re-training the model with recent data allows for easily adapting the approach to evolving crash traces, e.g., when the popularity of programming languages and APIs changes over time.

The model addresses the following problem:

Definition 3.1 (Line prediction problem). Given a crash trace $t = (l_1, ..., l_n)$ that consists of n lines, predict a vector of relevance scores $r = (s_1, ..., s_n)$, such that lines with a higher relevance score contain more information about the bug location.

Algorithm 1 Gathering training data for the trace-line model.

Input: Trace t, changeset C
Output: Vector r of relevance scores

```
r ← new vector
for all line l in t do
  s_max ← 0
  T ← tokenize(l)
  for all path p in C do
    P ← tokenize(p)
    s ← (P ∩ T) / P
    if s > s_max then
      s_max = s
  Append s_max to r
```

The remainder of this section presents how to gather data to train a supervised model that addresses the above problem (Section 3.1) and a neural network architecture we use to learn the model (Section 3.2).

3.1 Obtaining Historical Training Data

We address the line prediction problem through supervised machine learning, i.e., by learning from crash traces labeled with their most relevant lines. Scaffle creates this data by gathering pairs of crashes and fixes in the history of the code base and by computing a projected ground truth from these pairs. Each data point in the resulting training data is a pair (t, r) of a crash trace t and its corresponding vector r of relevance scores. Intuitively, r assigns those lines the highest relevance that point to (or that at least resemble) the paths in the code base where the crash was fixed.

To obtain crash-relevance pairs (t, r), we use crash traces associated with code changes that fix the root cause of the crash. For gathering crash traces and their associated changesets at Facebook, we perform an approach similar to past work that extracted such data from open-source repositories [30]. Some crashes are associated with issues that track progress toward fixing the underlying bug. Once fixed, the issue refers to the changeset that implements the fix. We gather pairs of crashes and changesets by combining both associations.

Given a pair (t, c) of a crash trace t and a changeset c, Scaffle obtains a crash-relevance pair (t, r) by comparing the paths affected by c with the lines in t. Algorithm 1 summarizes this step. It performs a pairwise comparison of each line in the trace and each file path affected by the changeset, which yields the relevance of the line for predicting the path. For this comparison, Scaffle tokenizes file paths into individual path segments and lines into individual words. Our tokenization function for trace lines does not assume any particular structure, but simply splits lines at every non-alphabetic character. Given a tokenized file path and a tokenized line, the algorithm computes the percentage of words in the path that are also contained in the line. Finally, the best score of a line across all file paths in the changeset is added to the relevance vector.

For illustration, reconsider the example in Figure 2(a) and suppose that it is part of the data that the trace-line model is learned from. Line 27 is tokenized into a sequence of words ("at", "dostuff", "DoStuffBrowserController", "clearBrowserFragment", "java", "775"). Suppose that the changeset consists of a single path "projectX/-packageA/dostuff/DoStuffBrowserController.java", which we tokenize into ("projectX", "packageA", "dostuff", "DoStuffBrowser-Controller", "java"). Because line 27 shares 3 out of 5 words with the path, the relevance score of the line is set to $\frac{3}{5} = 60\%$. Other lines contain less relevant information. For example, line 24 shares only the word "java" with the path, and therefore it is assigned a relevance score of $\frac{1}{5} = 20\%$. After computing the relevance score of each line, the algorithm concatenates all scores into a single relevance vector.

The algorithm for gathering labeled training data is a simple heuristic to identify the most relevant lines of a trace. In principle, the word-based matching of lines against file paths may not find the optimal relevance scores, and alternative definitions of Algorithm 1 exist. In practice, we find the algorithm to be an efficient way of producing training data that yields an effective trace-line model.

3.2 Neural Trace-Line Model

Scaffle uses the historical data, extracted as described above, as the ground truth for training a model that predicts the relevance of each line in a raw crash trace. In principle, different kinds of models could be trained for this purpose. We here present a neural network-based model, since neural networks have been proven to be highly effective at reasoning about raw input data, without the need to define and extract features of the input. The model reasons about a trace as a sequence of lines, each of which is a sequence of words. To tokenize a trace, we use the same tokenizer in Algorithm 1, which splits the text at every non-alphabetic character.

Figure 3 gives an overview of the neural network architecture. The network consists of two bi-directional recurrent neural networks (RNNs). The first RNN, called the *line-level RNN*, summarizes the words of each line into a continuous vector representation. The second RNN, called the *trace-level RNN*, takes a sequence of line-level vectors and predicts the relevance score of each line. Intuitively, this decomposition reflects how a human understands a trace, i.e., by understanding individual lines and by then reasoning about the meaning of multiple lines.

The input given to the trace-line model is a sequence of lines $t = (l_1, ...l_n)$, where each line consists of a sequence of words $l_i = (w_1, ..., w_k)$. To ease the training, we pad lines with fewer than k words and truncate lines that are longer than k words. By default, $k = 30$, which is sufficient to represent 98.4% of all lines in our dataset without truncating any words. Similarly, we fix the number of lines per trace to n by padding or truncating traces that are too short or long, respectively. By default, $n = 100$, which is shorter than most raw traces in our dataset, but covers most information relevant for localizing the buggy file. Each word is represented as a real-valued vector of length $e = 100$. To encode words into vectors, we pre-train Word2vec embeddings [16] on all traces. Intuitively, the embeddings assign a similar vector to words that occur in similar contexts.

The input to the line-level RNN is a matrix $\mathbb{R}^{k \times e}$. We use a bi-directional RNN that summarizes the sequence of words both in a forward and a backward pass and then concatenates both summaries into a single vector. The line-level RNN outputs a vector of length $\gamma = 140$, which summarizes the content of the line. Given

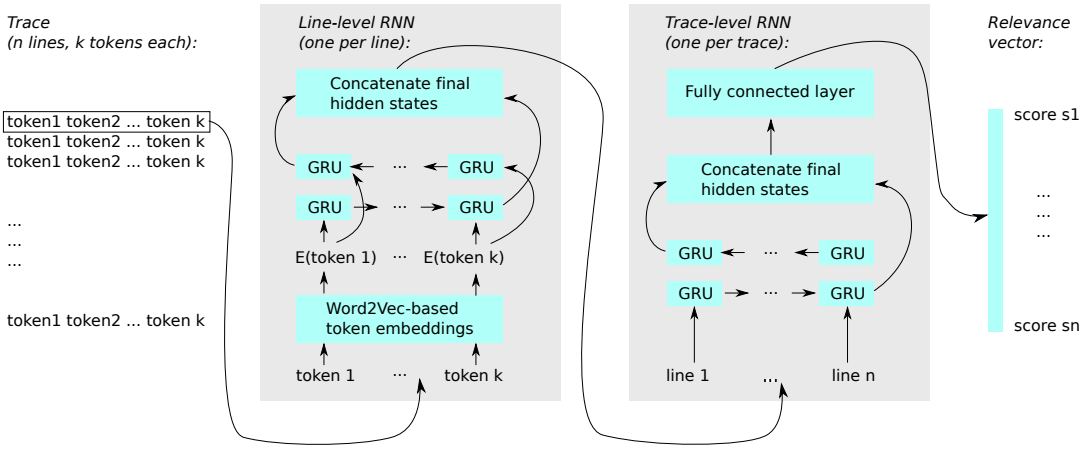


Figure 3: Neural trace-line model.

the continuous vector representation of each line, the trace-level RNN takes the sequence of lines encoded in a matrix $\mathbb{R}^{l \times \gamma}$. Similar to the line-level RNN, the trace-level RNN feeds the given sequence through a bi-directional RNN followed by a fully-connected layer. Finally, the model predicts the relevance vector $r \in \mathbb{R}^n$.

All parts of the neural trace-line model are trained jointly based on the ground truth of trace-relevance pairs. The rationale is to allow the network to find a representation of words and lines that is most suitable for addressing the line prediction problem.

4 PREDICTING PATHS FROM TRACE LINES

The second sub-problem that Scaffle addresses is to match the most relevant line of a crash trace, as predicted by the trace-line model, to file paths in the code base. This sub-problem comes with two main challenges. One challenge is to scale well to code bases that contain millions of files. Scaffle partially addresses this challenge by focusing on a single line of a trace, instead of comparing each line with the code base. Another challenge is that the most relevant trace line may not mention the exact file path where the bug is located. Possible reasons are that a prefix of a file path may be missing because the crash trace uses a path relative to the crash location, or that the trace simply does not contain the file name.

Our approach to predict file paths from trace lines addresses the above challenges by formulating the problem as an information retrieval (IR) problem. While IR is usually concerned about retrieving a few out of many documents for a given query, we aim at retrieving a few out of many file paths for a given trace line. The second part of Scaffle addresses the following problem:

Definition 4.1 (Path prediction problem). Given a trace line l and a set P of file paths in a code base, predict a ranked list $(p_1, p_2, \ldots)$ of file paths, with $p_i \in P$, such that l is most likely to refer to the highest ranked paths.

Inspired by IR techniques that retrieve documents for a given natural language query, Scaffle represents both the trace line (i.e., query) and each file path in the code base (i.e., document) as a set of words. File paths are tokenized into words by path segments, and in a similar manner, the trace line is tokenized by punctuation after stripping line numbers. An IR-based search engine is then run on the file paths to index the words appearing in them. Scaffle allows this search engine to be generic – in our evaluation (Section 6), we used a search tool based on vector space embedding, similar to [20]. However, any off-the-shelf search tool could be plugged into the approach. For instance, we also experimented with the Okapi BM25 function, which is common in elastic search, and found the difference between the two to be negligible.

When indexing the corpus of file paths, most modern search engines down-weight words that are prevalent across the corpus, and up-weight words that are more distinctive. For the example in Figure 2(a), the query consist of the words "at dostuff DoStuffBrowserController exitDoMoreStuff DoStuffBrowserController java". Because "java" is a very common word, it gets down-weighted, while "DoStuffBrowserController" is more distinctive and hence up-weighted. The IR component used in Scaffle weights words based on their tf-idf weight.

Once the search engine has indexed the corpus of file paths, Scaffle queries it with words from the most relevant lines predicted by the trace-line model. In the end, the result of path prediction is a list of paths $p_1, p_2, \ldots$ returned from the search query, ranked by IR-based similarity of the paths to the query.

An alternative to our IR-based way of addressing the path prediction problem would be to simply return the best-matching file based on the number of overlapping tokens. However, this alternative approach would be misled by commonly occurring terms, such as underlying framework and library names. An IR-based search overcomes this problem (and a few others) by weighting terms up/down depending on their distinctiveness in the corpus.

5 IMPLEMENTATION

We implement the trace-line model in PyTorch using gensim's implementation of Word2vec. The Word2vec embedding layer contains 100 dimensions, the line-level RNN consists of two hidden layers with 70 GRU cells each, and the trace-level RNN has two layers of 250 GRU cells each. We use the sigmoid function for activation and mean-squared error as the loss function. The model is trained for 50 epochs using the Adam optimizer with a learning rate of 0.001. The path prediction part of Scaffle is implemented using the standard tf-idf vectorizer in scikit-learn.

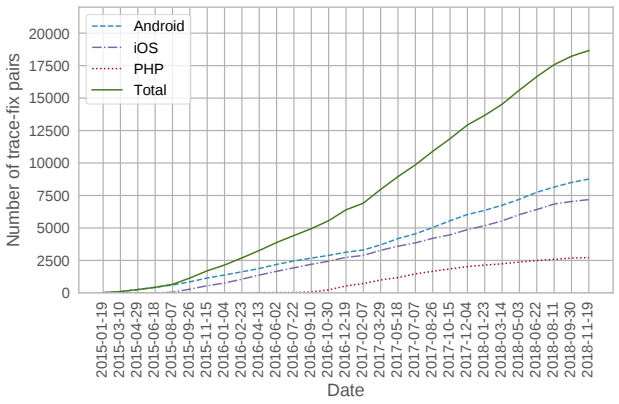


Figure 4: Number of crash traces in our dataset.

6 EVALUATION

We evaluate Scaffle with tens of thousands of crash traces and a code base consisting of millions of files in multiple programming languages. We address the following research questions:

- RQ 1: How effective is Scaffle at predicting bug locations?
- RQ 2: Given a raw trace, how effective is the learned trace-line model at identifying the most relevant lines?
- RQ 3: Why does Scaffle (sometimes not) work?
- RQ 4: How does Scaffle compare to existing approaches and to simpler baseline approaches?
- RQ 5: Is Scaffle efficient enough to scale to large code bases?

6.1 Experimental Setup

Our evaluation is based on tens of thousands of field crashes and their corresponding bug fixes. The data has been selected among crashes at Facebook over several years and comprises crashes from a diverse set of products. Field crashes observed in these products are automatically clustered to avoid inspecting the same problem multiple times. We use at most one representative of each such cluster, i.e., crashes in our dataset are unique. The crashes consist of Android crashes, such as the example in Figure 2(a), iOS crashes, and crashes in PHP code, such as the example in Figure 2(b), which contribute 46.8%, 38.7%, and 14.5% to the dataset, respectively. For each crash, we have a raw crash trace, which we feed into Scaffle without any preprocessing or parsing, except splitting the trace into lines. To establish ground truth data for bug localization, we associate crashes with bug fixing commits based on an issue tracker-like system at Facebook. Each bug fix changes one or more files, which we consider to be the bug location to predict. The minimum, mean, and maximum number of files changed in a bug-fixing commit is 1, 1.8, and 512, respectively.

Compared to similar setups in the literature [9, 12, 13, 18, 19, 21, 25, 31–33], there are two important differences: First, our dataset contains data from multiple projects and programming languages, which results in a more diverse set of crash traces. Second, our code base contains millions of files, i.e., it is at least two orders of magnitude larger than the largest previously considered code base.

To evaluate Scaffle in a realistic setup, we simulate using the approach at different points in time. Figure 4 shows the cumulative number of crash traces used in the evaluation. At each point in time t shown on the horizontal axis, we simulate using Scaffle by training its model based on all data available at t and by predicting the bug locations for all crash traces that occur between t and $t + 50$ days. For the prediction, we gather the set of all files in the code base at $t + 50$ days and let the path prediction component of Scaffle predict which of these files need to be fixed. This setup is realistic, as it uses only past data to predict future bug locations. A possible, but less realistic alternative would be to randomly split all available data into a training and a validation set.

We evaluate Scaffle on two variants of the crash traces in our dataset: raw traces and stack traces. *Raw traces* contain the crash stack, additional telemetry, and information added by other tools that handle crashes at Facebook. The examples in Figure 2 are raw traces. Particularly, raw traces contain the output of a heuristic logic that is used to aggregate crash reports into groups. This output is then used by engineers to identify the files of the codebase relevant to the crash, and so it serves as a *de facto* bug localization method. The goal here is to evaluate whether in an industrial setting with additional information as in raw traces, Scaffle can still add value. We compare Scaffle with the heuristic logic alone in Section 6.5. *Stack traces* contain only the crash stack that we extracted from the raw traces, and are stripped of any other information. In Figure 2(a) and (b), the extracted stack traces begin at lines 2 and 31, respectively. The goal here is to evaluate Scaffle in a more "pure" setting where only crash stacks are available [25, 30, 33].

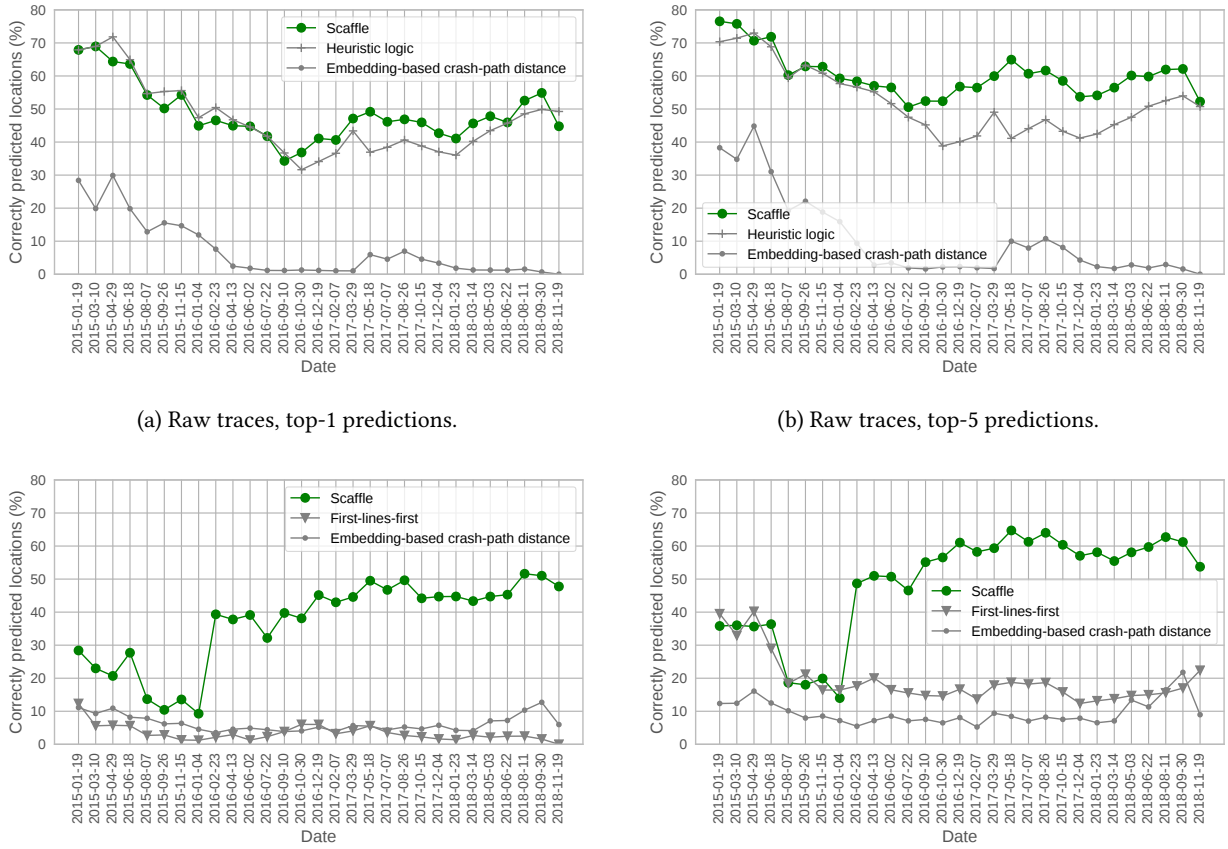
6.2 RQ 1: End-to-End Effectiveness

To measure how effective Scaffle is at predicting bug locations, we ask the approach to predict a ranked list of likely buggy files and then compare these files to those that have actually been changed by the developers. If the top predicted files include any of the actually changed files, we consider the prediction to be correct. The rationale is that pinpointing at least one of the files to modify is helpful in practice to identify which developer should handle a crash and which part of the code base the developer should focus on.

Figure 5 shows the percentage of correct predictions among the top-n predicted files. The plots on the left and right are for top-1 and top-5 predictions, respectively. The plots at the top and bottom are for raw traces and stack traces, respectively. The results vary over time because at each time step, a different model gets trained and because the crashes used for the evaluation vary from one 50-day period to another. The gray lines show baselines that we discuss in more detail in Section 6.5.

Overall, Scaffle is effective at selecting the bug location from millions of possible files. The model generally predicts between 40% and 60% of all bug locations as the top-1 prediction, and between 50% and 70% in the top-5 predictions. The results change over time for mostly two reasons. First, the composition of crashes in our dataset changes over time, as illustrated in Figure 4. The relative percentage of Android crashes decreases, while the percentage of crashes in iOS and PHP increases. In particular, it is only at some point in 2016 that the dataset starts to contain PHP crashes.[1] Second, for stack traces the model seems to require a certain number of training examples to unfold its full power. As evidenced by the steep

[1]The composition of crashes is not representative for software or crashes at Facebook, but merely a result of our data gathering process.



(a) Raw traces, top-1 predictions.

(b) Raw traces, top-5 predictions.

(c) Stack traces, top-1 predictions.

(d) Stack traces, top-5 predictions.

Figure 5: Top-n accuracy of predicted bug locations. The green line is the Scaffle approach.

increase in accuracy toward the beginning of 2016 in Figure 5(c) and (d), the model is much more effective once it has seen enough data.

6.3 RQ 2: Effectiveness of Trace-Line Model

To better understand Scaffle, we evaluate how effective the trace-line model is at predicting the most relevant lines in a crash trace. We use two metrics: Hit rate at n (hit@n) and mean reciprocal rank (MRR). Both metrics are computed w.r.t. the most relevant lines, as defined in the ground truth computed with Algorithm 1. The hit@n metric is the percentage of traces where the most-relevant line is among the top-n lines predicted by the model. For example, if the most relevant line is predicted as the second-most relevant line by the model, then this counts as a hit@3 but not as a hit@1. The MRR metric is computed by taking the predicted rank of the most relevant line, computing its reciprocal, and then averaging across all traces. For example, suppose there are only two traces and that the most relevant line of trace 1 and trace 2 is predicted at rank 1 and 4, respectively, then the MRR is $\frac{1}{2} \cdot (\frac{1}{1} + \frac{1}{4}) = 0.625$.

Figure 6 shows the results for the hit@n metric. Again, the green lines are for the Scaffle approach; all other lines are baselines discussed in Section 6.5. Solid lines show hit@1 results, dotted lines show the corresponding hit@5 results. The results roughly follow the same patterns as the end-to-end results in Figure 5, confirming that the effectiveness of the trace-line model is key to the overall success of Scaffle. Notably, the model achieves a hit@5 rate above 80% for most points in time.

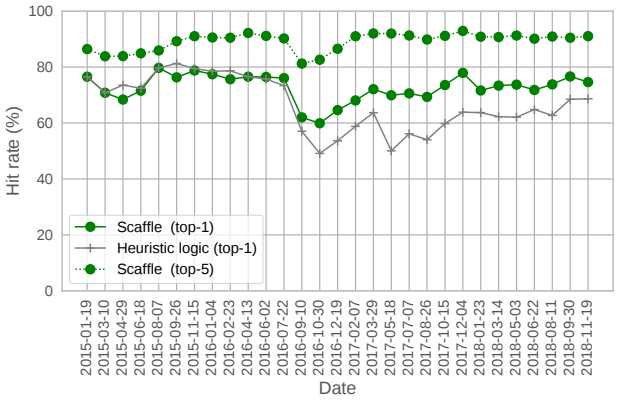
Figure 7 shows the results for the MRR metric. Again, we find the trace-line models to be highly effective, reaching MRR values of over 0.8 for raw traces and close to 0.8 for stack traces.

6.4 RQ 3: Why Does Scaffle Work?

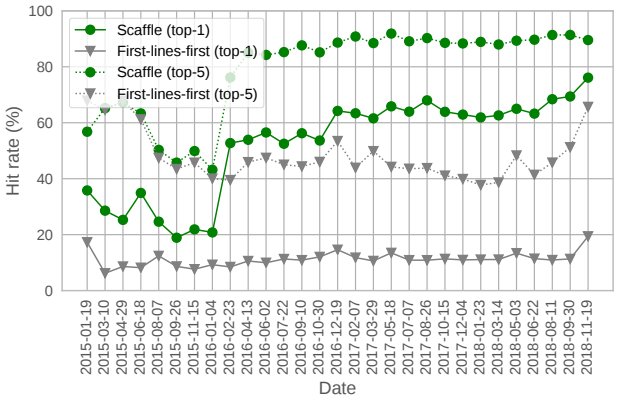
To better understand why the approach often is, and sometimes is not, able to identify the files relevant for fixing a bug, we manually inspect various traces and their corresponding bug locations. This inspection leads to the following observations.

Files mentioned in traces. Perhaps unsurprisingly, Scaffle is effective when the file that needs to be fixed is mentioned in the crash trace, e.g., because one of the functions in the file appears in a stack trace mentioned in the trace. In contrast, the approach cannot predict the correct bug location when the buggy file is not mentioned anywhere in the trace. The latter case may happen when the root cause and the manifestation of a bug are in different files.

Partial information about files. Even incomplete mentions of a file may be sufficient to enable Scaffle to localize it. For example, some crash traces mention the relevant file name, but not the complete

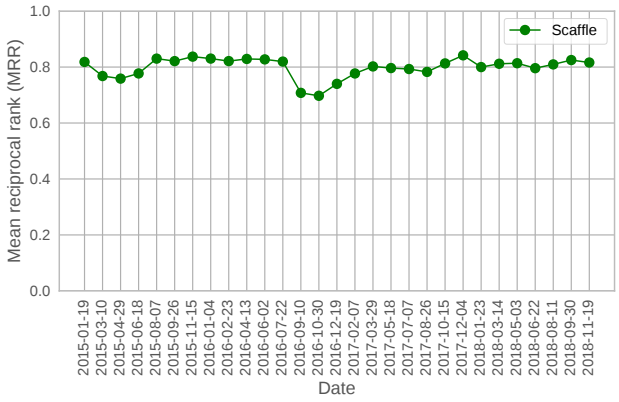


(a) Raw traces.

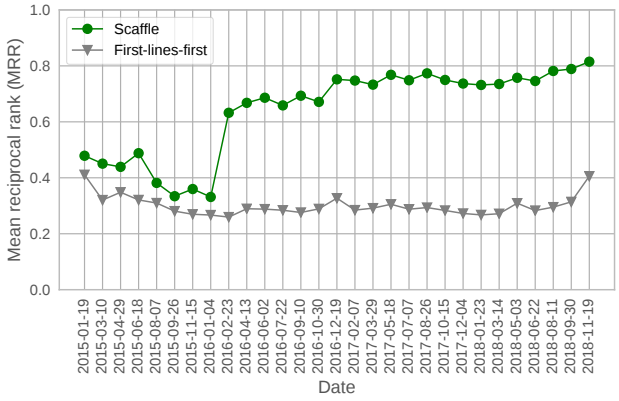


(b) Stack traces.

Figure 6: Hit rate within the top-n predictions of most relevant lines. The green lines are for the Scaffle approach.



(a) Raw traces.



(b) Stack traces.

Figure 7: MRR of the trace-line model and a heuristic baseline. The green lines are for the Scaffle approach.

path of the file. The reason may be that file paths on deployment devices differ from those in the code base or that the crash trace format does not include complete paths. Scaffle's IR-based matching of lines and file paths exploits the fact that some high-entropy segments of a path, e.g., only the file name, often are sufficient to uniquely identify a path. In contrast, the approach sometimes fails to uniquely identify the correct file because multiple files with the same file name exist in different directories of the code base.

Understanding the structure of traces. The neural trace-line model has some "understanding" of the structure of raw crash traces. As illustrated by the examples in Figure 2, it identifies stack traces within the raw traces by giving lines that are part of stack trace generally higher relevance scores. Moreover, the model learns to identify the relevant lines within a stack trace by discarding stack frames unlikely to point to a bug location. For example, the model gives a relatively low score to those lines in Figure 2(a) that start with `at android.app`, because they refer to methods in the Android framework, i.e., code unlikely to cause a bug in the application. The model also learns to handle nested stack traces, such as the one in Figure 2(a), where one exception causes another. For such stack traces, the model learns to search the most relevant lines in the inner-most exception, as this exception is the cause of the crash.

Instead of learning to understand raw traces and the stack traces contained in them, one could manually implement heuristic algorithms for parsing different trace formats. One of our baselines (Section 6.5) are a set of such heuristics. The main benefit of a learned model is that it can be obtained automatically and that it can be easily re-trained when trace formats evolve.

6.5 RQ 4: Comparison with Prior Work and Baselines

6.5.1 End-to-End Classification. Scaffle breaks down bug localization into two phases: predicting the most relevant lines in a crash trace and retrieving the most relevant files based on these lines. Prior work, e.g., by Kim et al. [9], proposes to directly predict the buggy file from a bug report or crash trace. The basic idea is to extract features of the crash trace, such as words appearing in the report, and train a classifier on past fixes to predict a set of relevant files to fix. To improve the precision of their model, they split the process into two phases: one that predicts if a crash report contains useful information, and one that actually predicts the file.

We implement the model described by Kim et al. [9] using a Naive Bayes and a Random Forest classifier. Unfortunately, the model suffers from acute scalability problems, and we were unable to train it on more than 10,000 files before running out of memory. The reason is the large number of file paths, each of which is a unique "class" for the classifier. Generally, classifiers scale well with the number of features, but not with the number of classes. Since an end-to-end classification approach, such as [9], considers each file as a class, it does not scale to code bases with millions of files.

6.5.2 End-to-End Information Retrieval. IR-based bug localization is one of the most prevalent approaches proposed in prior work [13, 17, 19, 21, 27, 33]. We compare Scaffle with a scalable variant of these approaches that avoids running a source-level analysis of all files in the code base. This end-to-end, IR-based approach considers the file paths as documents, similar to Scaffle, and matches them against the given crash trace. To implement this baseline, we use the path prediction model in Section 4 to embed file paths into a tf-idf based vector space. Given a crash trace, the baseline embeds the entire trace into the same vector space, and reports those file paths as most relevant for the crash that have the lowest cosine distance to the embedded crash trace.

The result of this baseline is plotted in Figure 5 ('Embedding-based crash-path distance"). The baseline performs poorly compared to Scaffle, especially as the number of crashes and file paths grow over time. The reason is that the efficiency of IR techniques relies largely on the quality of the query provided for search. However, if the query is the entire crash trace, it contains significant amounts of noise, which adversely affects the accuracy of retrieval. The comparison with this baseline shows that decomposing the bug localization problem by first predicting the most relevant line of a crash trace is key to achieving good overall accuracy.

6.5.3 First Lines First. A simple baseline to identify the most relevant lines in a given stack trace is to assume that the lines in the stack trace are sorted by descending relevance. This baseline, which we call *first-lines-first*, matches an assumption made in prior work on bug localization [19]. All figures for stack traces show the results for this baseline as a gray line, i.e., Figure 5(c) and (d), Figure 6(b), and Figure 7(b). We find the first-lines-first baseline to perform clearly worse than Scaffle because the root cause of a crash is not always mentioned in one of the first lines.

6.5.4 Heuristic Logic. As an industrially deployed baseline, we consider a heuristic logic that aggregates crash reports at Facebook. It is based on a series of manually designed and tuned rules that extract relevant strings from the crash, such as the exception message, type, or parts of stack frames. The output is then added to the raw trace and is used to group together crashes that have the same output from the logic. Engineers use it to gain hints about files that are relevant to the crash. To measure whether Scaffle adds any value on top of the heuristic logic, we consider the heuristic logic as a stand-alone baseline and show its effectiveness in all figures for raw traces as a gray line, i.e., in Figure 5(a) and (b), and in Figure 6(a). For the end-to-end prediction, we find this baseline to be less effectiveness than Scaffle. In other words, the model learns to override the suggestions of the heuristic logic by predicting other lines in a raw trace than the result of the logic as the most relevant. This shows that even in an industrial setting, Scaffle can be used to augment any existing localization methods. Another advantage of Scaffle is to predict a ranked list of likely relevant lines and likely buggy files. For example, as shown in Figure 6(a), the top-5 predictions have a higher hit rate than the heuristic baseline.

6.6 RQ 5: Efficiency

To evaluate the efficiency of Scaffle, we distinguish between one-time efforts and per-trace efforts. The computationally most expensive one-time effort is to train the trace-line model. Depending on the amount of training data, the training takes up to three hours on a single machine equipped with a standard GPU. The time to predict the bug location for a given crash trace is the sum of the time needed by the two main steps of Scaffle. Querying the model with a given trace generally takes less than a second. The IR-based matching of the top-most predicted lines against all file paths in the code base takes up to several seconds per line. This amount of time is acceptable in practice because Scaffle can run in the background and report its suggestions to developers once it is done. Overall, we conclude that Scaffle is efficient and scalable enough to run in an industrial deployment, even for large-scale code bases.

7 DISCUSSION

This section discusses some limitations and alternative designs of the approach. One limitation is that Scaffle predicts only one file out of possibly many files that cause a crash. On average over all crashes used in the evaluation, we find that 1.8 files are modified to fix a crash. That is, finding one of the relevant files covers a large fraction of all relevant files in practice. Even if multiple files need to be modified, predicting one of them still provides a good starting point to find the right team or developer to handle the crash, and to find the remaining files using other techniques.

The prediction accuracy of Scaffle ranges between 40% and 70%, which raises the question whether it is high enough to be useful in practice. While a higher accuracy would certainly be desirable, the current result can contribute some value. One piece of evidence is that Scaffle outperforms the heuristic logic that is currently used at Facebook to help developers handle crashes (Section 6.5.4). Given that crash-based bug localization is a hard yet practically relevant problem, we envision future work to further improve the current result. One promising direction may be to combine a scalable technique, such as Scaffle, with a more expensive technique that reasons only about selected subsets of the code base.

Our evaluation learns a single trace-line model across all crash traces, irrespective of the specific product that has crashed or the programming language used to write the crashing code. Instead, one could train a separate model for different subsets of our dataset. There are at least two reasons why we choose a single-model approach. One reason is that the crash traces contain language-independent information added by tools that pre-process traces at Facebook. By learning a single model across all crashes, the model can generalize better and learn patterns that hold across products and languages. Another reason is that the set of products, programming languages, etc. is constantly evolving. A single model eliminates the need to build multiple pipelines and is more robust toward an evolving code base.

Scaffle is designed with ultra-large scale, heterogeneous code bases in mind. Smaller software projects may not have any or not enough information about past crashes available. For such projects, our approach may not work well, because training an effective trace-line model requires a sufficiently large training dataset. For such smaller projects, the code base may be small enough to analyze each source file individually, making other existing techniques (Table 1) a viable alternative.

8 RELATED WORK

Localizing Bug Locations. The most closely related line of work also predicts bug locations based on some evidence of a bug, such as a bug report or a stack trace, as summarized in Table 1. Our work, shown in the last row, differs by taking raw crash traces as the input, without making any assumptions about their format. Almost all other techniques also differ from ours by focusing on a single programming language and by requiring some form of parsing and code analysis of all files in the code repository. The work by Wang et al. [25] is the exception, as it also targets multiple languages and does not require parsing or analyzing the code. It differs from Scaffle in multiple ways: First, their approach learns a model that, given features of a specific file, predicts the probability that the file contains the bug. This design implies that predicting the buggy file requires querying the model with all files in the code base, which does not scale well to code bases with millions of files, such as the one that motivates our work. Second, their approach assumes that file names mentioned in a crash trace can always be resolved to file paths in the code base. We find this assumption to sometimes be violated, e.g., because the path of a file on a client device differs from the path in the code base. Third, their approach expects a set of stack traces as the input, whereas Scaffle works well given only a single crash trace. Some of the work listed in Table 1 uses inputs beyond bug reports or crash traces, e.g., by also considering meta-data from version histories, such as files involved in fixing past bugs [13, 32]. Scaffle does not require this kind of meta-data, but could possibly benefit from it.

Localizing Bug-Inducing Changes. A related problem is to localize which commit is causing a bug. Locus addresses this problem in an IR-based approach that compares the words associated with a commit and the words in a bug report [26]. Orca also takes an IR-based approach and reports deploying a system for a large-scale, distributed, industrial system [2]. Their focus is on handling the frequent re-builds of the system through a build provenance graph. ChangeLocator takes a learning-based approach that ranks all commits that change at least one method that appears in a stack trace [29]. Similar to most techniques in Table 1, ChangeLocator relies on statically analyzing the code base, which Scaffle avoids.

Clustering and Prioritizing Crashes. The potentially large number of crashes revealed by fuzz-testing or widely deploying software has motivated work on clustering crashes, e.g., based on similarities of stack traces [4, 5], on repairs that prevent a crash [23], or other heuristics [11]. Once crashes are clustered, Castelluccio et al. [3] propose to help understanding the crash by identifying features that are unique to a cluster. Another line of work prioritizes crashes, e.g., based on the distribution of occurrences among users [8] or based on a prediction of how likely a crash will occur for other users [10]. All this work is orthogonal to the problem addressed here, and could be used before localizing the bug location for a crash with Scaffle.

Other Related Work. Many techniques for automated program repair [14] rely on localizing where to fix a bug. Scaffle could serve as a starting point for repair of bugs that manifest through crashes. The broader problem of bug localization has received significant attention. We here restrict our discussion to crash-based localization and refer to a survey [28] for detailed discussion. The main difference to coverage-based bug localization [1, 6, 15] is that Scaffle does not require any coverage information, but only a crash trace that manifests the bug. Wang et al. [24] study whether IR-based bug localization for given bug reports simplifies debugging. Our setup differs by considering machine-generated, raw crash traces instead of (at least partially) human-written bug reports. Our trace-line model exploits one of their observations, that "some stack traces and test cases contain many class names and method names, and only a small subset of the names are closely related to the bug" [24]. Jonsson et al. [7] address the problem of assigning a bug report to a developer team. They also focus on large-scale deployment, but with dozens of teams instead of millions of files to choose from.

9 CONCLUSIONS

This paper presents Scaffle, the first technique for automated, crash-based bug localization in ultra-large scale, heterogeneous code bases. The key idea is to decompose the problem into two simpler sub-problems: (1) Identifying the most relevant lines in a raw crash trace, which we address through a neural trace-line model, and (2) Matching these lines with file paths in the code base, which we address through a scalable, IR-based search. The approach is language-independent, as it does not assume a specific trace format, but instead learns from crashes fixed in the past. Our evaluation applies Scaffle to a code base that is at least two orders of magnitude larger than any previous work. We find the approach to be effective at identifying files to fix, despite having to choose among millions of code files written in several programming languages. Notably, Scaffle outperforms several non-trivial baselines, including end-to-end classification, an end-to-end IR-based search, and industrially used heuristics. Overall, Scaffle provides a practical technique for pinpointing the files to consider in a large-scale code base, which helps assigning crashes to the appropriate team or developer, and may serve as a starting point for automated program repair.

ACKNOWLEDGMENTS

Parts of this work were supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 851895) and by the German Research Foundation within the ConcSys and Perf4JS projects.

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 89–98.

[2] Ranjita Bhagwan, Rahul Kumar, Chandra Shekhar Maddila, and Adithya Abraham Philip. 2018. Orca: Differential Bug Localization in Large-Scale Services. In

13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018. 493–509. https://www.usenix.org/conference/osdi18/presentation/bhagwan

[3] Marco Castelluccio, Carlo Sansone, Luisa Verdoliva, and Giovanni Poggi. 2017. Automatically analyzing groups of crashes for finding correlations. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017.* 717–726. https://doi.org/10.1145/3106237.3106306

[4] Yingnong Dang, Rongxin Wu, Hongyu Zhang, Dongmei Zhang, and Peter Nobel. 2012. ReBucket: A method for clustering duplicate crash reports based on call stack similarity. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland.* 1084–1093. https://doi.org/10.1109/ICSE.2012.6227111

[5] Tejinder Dhaliwal, Foutse Khomh, and Ying Zou. 2011. Classifying field crash reports for fixing bugs: A case study of Mozilla Firefox. In *IEEE 27th International Conference on Software Maintenance, ICSM 2011, Williamsburg, VA, USA, September 25-30, 2011.* 333–342. https://doi.org/10.1109/ICSM.2011.6080800

[6] James A. Jones, Mary Jean Harrold, and John T. Stasko. 2002. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA.* 467–477. https://doi.org/10.1145/581339.581397

[7] Leif Jonsson, Markus Borg, David Broman, Kristian Sandahl, Sigrid Eldh, and Per Runeson. 2016. Automated bug assignment: Ensemble-based machine learning in large scale industrial contexts. *Empirical Software Engineering* 21, 4 (2016), 1533–1578. https://doi.org/10.1007/s10664-015-9401-9

[8] Foutse Khomh, Brian Chan, Ying Zou, and Ahmed E. Hassan. 2011. An Entropy Evaluation Approach for Triaging Field Crashes: A Case Study of Mozilla Firefox. In *18th Working Conference on Reverse Engineering, WCRE 2011, Limerick, Ireland, October 17-20, 2011.* 261–270. https://doi.org/10.1109/WCRE.2011.39

[9] Dongsun Kim, Yida Tao, Sunghun Kim, and Andreas Zeller. 2013. Where Should We Fix This Bug? A Two-Phase Recommendation Model. *IEEE Trans. Software Eng.* 39, 11 (2013), 1597–1610. https://doi.org/10.1109/TSE.2013.24

[10] Dongsun Kim, Xinming Wang, Sunghun Kim, Andreas Zeller, Shing-Chi Cheung, and Sooyong Park. 2011. Which Crashes Should I Fix First?: Predicting Top Crashes at an Early Stage to Prioritize Debugging Efforts. *IEEE Trans. Software Eng.* 37, 3 (2011), 430–447. https://doi.org/10.1109/TSE.2011.20

[11] Kinshuman Kinshumann, Kirk Glerum, Steve Greenberg, Gabriel Aul, Vince R. Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen C. Hunt. 2011. Debugging in the (very) large: ten years of implementation and experience. *Commun. ACM* 54, 7 (2011), 111–116. https://doi.org/10.1145/1965724.1965749

[12] Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Kui Liu, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2019. D&C: A Divide-and-Conquer Approach to IR-based Bug Localization. *IEEE Transactions on Software Engineering* (2019).

[13] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2015. Combining Deep Learning with Information Retrieval to Localize Buggy Files for Bug Reports (N). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015.* 476–481. https://doi.org/10.1109/ASE.2015.73

[14] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65. https://doi.org/10.1145/3318162

[15] Xiangyu Li, Shaowei Zhu, Marcelo d'Amorim, and Alessandro Orso. 2018. Enlightened debugging. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018.* 82–92. https://doi.org/10.1145/3180155.3180242

[16] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States.* 3111–3119.

[17] Laura Moreno, John Joseph Treadway, Andrian Marcus, and Wuwei Shen. 2014. On the Use of Stack Traces to Improve Text Retrieval-Based Bug Localization. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014.* 151–160. https://doi.org/10.1109/ICSME.2014.37

[18] Anh Tuan Nguyen, Tung Thanh Nguyen, Jafar M. Al-Kofahi, Hung Viet Nguyen, and Tien N. Nguyen. 2011. A topic-based approach for narrowing the search space of buggy files from a bug report. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011.* 263–272. https://doi.org/10.1109/ASE.2011.6100062

[19] Mohammad Masudur Rahman and Chanchal K. Roy. 2018. Improving IR-based bug localization with context-aware query reformulation. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018.* 621–632. https://doi.org/10.1145/3236024.3236065

[20] Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. 2018. Retrieval on source code: a neural code search. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages.* ACM, 31–41.

[21] Ripon K. Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E. Perry. 2013. Improving bug localization using structured information retrieval. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013.* 345–355. https://doi.org/10.1109/ASE.2013.6693093

[22] Mozhan Soltani, Annibale Panichella, and Arie van Deursen. 2017. A guided genetic algorithm for automated crash reproduction. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017.* 209–220. https://doi.org/10.1109/ICSE.2017.27

[23] Rijnard van Tonder, John Kotheimer, and Claire Le Goues. 2018. Semantic crash bucketing. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018.* 612–622. https://doi.org/10.1145/3238147.3238200

[24] Qianqian Wang, Chris Parnin, and Alessandro Orso. 2015. Evaluating the usefulness of IR-based fault localization techniques. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015.* 1–11. https://doi.org/10.1145/2771783.2771797

[25] Shaohua Wang, Foutse Khomh, and Ying Zou. 2013. Improving bug localization using correlations in crash reports. In *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013.* 247–256. https://doi.org/10.1109/MSR.2013.6624036

[26] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: locating bugs from software changes. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016.* 262–273. https://doi.org/10.1145/2970276.2970359

[27] Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, and Hong Mei. 2014. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *2014 IEEE International Conference on Software Maintenance and Evolution.* IEEE, 181–190.

[28] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Trans. Software Eng.* 42, 8 (2016), 707–740. https://doi.org/10.1109/TSE.2016.2521368

[29] Rongxin Wu, Ming Wen, Shing-Chi Cheung, and Hongyu Zhang. 2018. ChangeLocator: locate crash-inducing changes based on crash reports. *Empirical Software Engineering* 23, 5 (2018), 2866–2900. https://doi.org/10.1007/s10664-017-9567-4

[30] Rongxin Wu, Hongyu Zhang, Shing-Chi Cheung, and Sunghun Kim. 2014. CrashLocator: locating crashing faults based on crash stacks. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014.* 204–214. https://doi.org/10.1145/2610384.2610386

[31] Xin Ye, Razvan Bunescu, and Chang Liu. 2015. Mapping bug reports to relevant files: A ranking model, a fine-grained benchmark, and feature evaluation. *IEEE Transactions on Software Engineering* 42, 4 (2015), 379–402.

[32] Xin Ye, Razvan C. Bunescu, and Chang Liu. 2014. Learning to rank relevant files for bug reports using domain knowledge. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014.* 689–699. https://doi.org/10.1145/2635868.2635874

[33] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland.* 14–24. https://doi.org/10.1109/ICSE.2012.6227210