DeepAnalyze: Learning to Localize Crashes at Scale

Manish Shetty
t-mamola@microsoft.com
Microsoft Research
Bangalore, India

Chetan Bansal
chetanb@microsoft.com
Microsoft Research
Redmond, USA

Suman Nath
sumann@microsoft.com
Microsoft Research
Redmond, USA

Sean Bowles
sbowl@microsoft.com
Microsoft
Redmond, USA

Henry Wang
hewang@microsoft.com
Microsoft
Redmond, USA

Ozgur Arman
oarman@microsoft.com
Microsoft
Redmond, USA

Siamak Ahari
sahari@microsoft.com
Microsoft
Redmond, USA

ABSTRACT

Crash localization, an important step in debugging crashes, is challenging when dealing with an extremely large number of diverse applications and platforms and underlying root causes. Large-scale error reporting systems, e.g., Windows Error Reporting (WER), commonly rely on manually developed rules and heuristics to localize *blamed frames* causing the crashes. As new applications and features are routinely introduced and existing applications are run under new environments, developing new rules and maintaining existing ones become extremely challenging.

We propose a data-driven solution to address the problem. We start with the first large-scale empirical study of 362*K* crashes and their blamed methods reported to WER by tens of thousands of applications running in the field. The analysis provides valuable insights on where and how the crashes happen and what methods to blame for the crashes. These insights enable us to develop DeepAnalyze, a novel multi-task sequence labeling approach for identifying blamed frames in stack traces. We evaluate our model with over a million real-world crashes from four popular Microsoft applications and show that DeepAnalyze, trained with crashes from one set of applications, not only accurately localizes crashes of the same applications, but also bootstrap crash localization for other applications with zero to very little additional training data.

ACM Reference Format:
Manish Shetty, Chetan Bansal, Suman Nath, Sean Bowles, Henry Wang, Ozgur Arman, and Siamak Ahari. 2022. DeepAnalyze: Learning to Localize Crashes at Scale. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3512759

1 INTRODUCTION

When software crashes in the wild, often the primary sources of information available for debugging are *crash stacks* – stack traces collected during the crashes [47]. A crash stack contains what methods were executing during a crash. It may also contain other valuable information such as executed binaries and code locations that can hint as to what might have caused the crash. Due to its importance, many error reporting systems, e.g., Windows Error Reporting (WER) [16], Apple Crash Reporter [4], Mozilla Crash Reporter [42], Chrome Crash Reporter [12], have been deployed to automate the collection of crash stacks (along with other information, such as memory dumps). An important step in investigating a crash is *crash localization*: identifying the method in the crash stack that contains, or is the closest to,[1] the crash location. We denote such a method as the *blamed method*, and the stack frame containing it as the *blamed frame*. Blamed methods play an important role in organizing crash reports into "buckets" (i.e., categories), which in turn help developers prioritize frequently seen buckets [16]. Moreover, investigation of a crash often starts from the blamed method as it helps developers isolate the crash location.

Crash localization needs to be automated in large-scale error reporting systems, such as WER, as they receive millions of crash reports per day [16]. A common practice [16, 53] is to use a collection of manually written heuristics such as "Never mark `Foo()` as a blamed method", "`Bar()` can be a blamed method only if the symbol `Baz` appears in the crash stack", and so on. The system then scans through the frames of a crash stack to identify a blamed method that is consistent with these rules. Ideally, the rules should be consistent (i.e., not contradictory) with each other, and have good accuracy and coverage. This is nontrivial for large and evolving systems; when a new feature or application is introduced, someone with good domain knowledge needs to write new application-specific rules. WER currently has 50+ such application-specific libraries of rules.

Prior work in crash analysis has heavily focused on crash bucketization [14, 16, 54]. But, in order to do effective bucketization, crash localization is critical. Wu et al. [59] proposed CrashLocator which leverages source code along with the static call graph for crash localization. However, this is not feasible at the scale of WER that needs to localize crashes for a multitude of applications in the wild. This paper addresses these limitations with a fresh data-driven approach. Inspired by the abundance of data in existing error reporting systems and recent advancement in deep learning techniques, we demonstrate how to learn from past crashes to identify the blamed method in a new crash stack with high accuracy.

[1] When a stack trace does not contain the true crash location [19], we consider identifying the method closest to the crash location in caller-callee relationship.

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3512759

0	msedge_elf.dll!crash_reporter::DumpWithoutCrashing
1	msedge.dll!base::debug::DumpWithoutCrashing
2	msedge.dll!gl::DirectCompositionChildSurfaceWin::ReleaseDrawTexture
3	msedge.dll!gl::DirectCompositionChildSurfaceWin::SwapBuffers
4	msedge.dll!gl::DirectCompositionChildSurfaceWin::SwapBuffers
5	msedge.dll!gl::GLSurfaceAdapter::PostSubBuffer
6	msedge.dll!gpu::PassThroughImageTransportSurface::PostSubBuffer
7	. . .

(a) Crash stack 1

0	igd10iumd64.dll!OpenAdapter10_2
1	d3d11.dll!NDXGI::CDevice::RotateResourceIdentities
2	dxgi.dll!CDXGISwapChain::PresentImplCore
3	dxgi.dll!CDXGISwapChain::PresentImpl
4	dxgi.dll!CDXGISwapChain::[IDXGISwapChain4]::Present1
5	msedge.dll!gl::DirectCompositionChildSurfaceWin::ReleaseDrawTexture
6	msedge.dll!gl::DirectCompositionChildSurfaceWin::SwapBuffers
7	. . .

(b) Crash stack 2

Figure 1: Examples of crash stacks from Microsoft Edge and their crash locations (red frame)

As a first step towards our data-driven approach, we analyze ≈ 362K crash stacks collected by WER from ≈ 8.7K software components. Our analysis highlights the huge diversity of crash stacks: they come from many different first and third party binaries, and from many different methods and namespaces within each binary. The underlying *problem classes*, which denote high level root cause types such as heap corruption or stack overflow, are also diverse. Finally, a crash trace often contains many methods, only one of which needs to be identified as the blamed method. All of the above highlight the challenges of manually developing and maintaining application-specific crash-localization heuristics.

Our analysis also provides several insights that guide our choice of an effective machine-learning solution. We first tried a linear binary classifier (Logistic Regression) that, given features of an individual frame, predicts the likelihood of it being the blamed frame. As we show in Section 5.1, this simple model, however, did not work well for many problem classes. Upon further analysis of our dataset, we found that *the context in which a method appears in a crash stack (e.g., methods that appear before and after it) plays an important role in it being a blamed method or not.* Consequently, a method can be the blamed method in one crash stack but not in another. This is illustrated with two crash traces from Microsoft Edge, shown in Figure 1. Both the traces contain Edge's method `ReleaseDrawTexture`, but WER identifies it as the blamed method only for the first trace. In the first trace, methods around `ReleaseDrawTexture` are relatively less crash-prone based on their past history (e.g., the logging methods above it). In the second trace, however, `ReleaseDrawTexture` has more crash-prone methods from a user mode Intel graphics driver, one of which is blamed by WER.

We use this insight on the importance of context in a novel formulation of crash localization as *sequence labeling*. Sequence labeling, widely used in natural language processing, uses context to assign a categorical label (e.g., parts of speech) to each member of a sequence. In our formulation, we treat a crash stack as a sequence of frames and aim to assign to each frame a binary category indicating whether it is a blamed frame or not. But, applying sequence labeling to crash localization requires addressing several challenges.

First, like many other machine learning tasks, we need to select a good set of features and a suitable model that can capture context. Here, we revisit our data analysis to seek insights. Our analysis shows that even though a stack trace may contain many methods, only a small number of them are likely to be a crash location, e.g., methods that appear towards the top of the stack, have global semantic importance, and that are implemented in application code rather than the underlying system. We therefore extract features

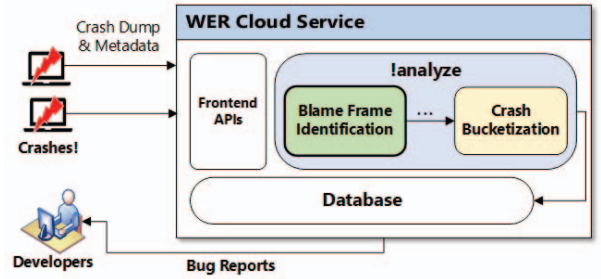


Figure 2: An overview of WER

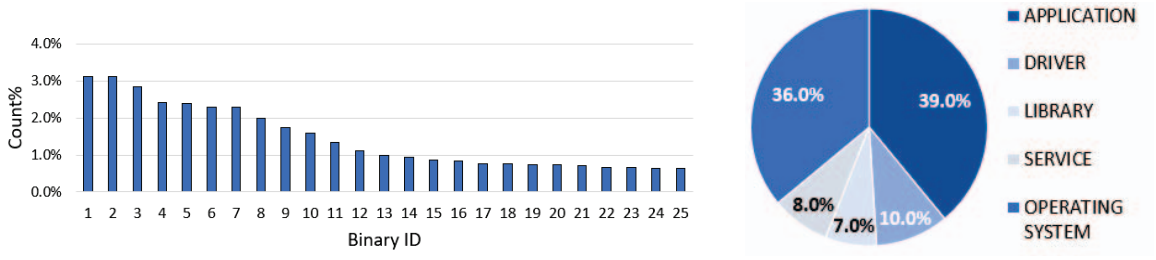
that summarize these properties of a stack frame. We also observe that context for a frame depends on the call chain through which it is invoked. We capture this sequential context flow in the stack using a Bi-directional Long Short-term Memory (Bi-LSTM) layer [21] that interprets the stack both top-down and bottom-up.

Second, traditional sequence labeling may label multiple tokens with the same category. In our context, however, only one frame in a stack trace can be blamed. To address this, we first use an attention mechanism [5] to identify sections of a stack trace that are more likely to contain the crash location. Then, we model the frame-level labeling task jointly using linear chain Conditional Random Fields (CRF) [27]. Finally, to tackle constantly evolving software, it is important that models learned from crashes in one set of applications are useful not only for those, but also for other applications, e.g., newly released ones that have very little training data. We propose a transfer learning approach to achieve this goal.

In this work, we present and package our models in a system called **DeepAnalyze** – a deep learning based solution for large-scale crash localization. We have evaluated DeepAnalyze with over a million real-world crash stacks from four popular Microsoft applications (Edge, Word, Excel, and Outlook). Our results show that DeepAnalyze's novel multi-task sequence labeling approach has an average accuracy of 0.9 and it outperforms several baselines. Also, we show that using transfer learning, our model, learned from one set of applications, can also be effectively (≈ 0.8 accuracy) used for completely new and unseen applications with no new training data. Lastly, we show that these models can be easily fine-tuned to new applications, where the accuracy quickly approaches ≈ 0.9 with as little as a few thousands additional training samples.

In summary, we make the following contributions:

(1) We conduct the first large-scale empirical study of crashes in the wild, and discuss new insights about crash sources, problem types and characteristics of their blamed methods.



(a) Top-25 crashing binaries **(b) Top-100 software component types**

Figure 3: Distribution of crashing binaries and software components

(2) We propose DeepAnalyze, a novel Multi-task learning based approach for crash localization only using stack traces.
(3) We evaluate DeepAnalyze on 4 popular Windows applications showing that it has an average accuracy of 0.9 and outperforms several baselines.
(4) We leverage transfer learning and demonstrate that with a small amount of data, we can localize crashes for new/unseen applications with high accuracy.

2 BACKGROUND

2.1 Windows Error Reporting

Released software often contains bugs that cause the software to crash in the field. To automate collecting crash information, large software companies deploy error reporting systems. For example, Microsoft has built a distributed error-reporting system called Windows Error Reporting (WER) [1, 16], which has been in operation for over two decades now. When a Microsoft software (such as Windows, Word, and Edge) crashes in the field, with user permission, it sends to WER a *crash report*. A crash report contains crash stacks, information of the crashed application and its runtime, and optionally a memory dump collected during the crash (Figure 2). So far, WER has collected tens of billions of crash reports from billions of devices.

A key functionality of WER is to assign *buckets* to the crashes so that similar bugs can de-duplicated and triaged together. Once the number of crashes in a bucket has exceeded a certain threshold, a bug report is created and it is triaged to the appropriate developer using the bucket metadata. A bucket is basically a signature to identify a unique bug. Here is an example of a bucket: *MEMORY_CORRUPTION_c0000005_contoso.exe!WriteToChild*. It contains the problem class, exception code and the blamed frame which caused the crash. The bucket is computed by analyzing a crash report, as described next.

2.2 Crash localization in WER

In order to analyze a crash report, WER uses `!analyze` [2], a debugger extension which uses the call stack from the crash dump along with metadata such as loaded modules, memory dump and exception code to identify blamed frame and the underlying problem class that caused the crash (e.g., out of memory, stack overflow). `!analyze` has been built and maintained for more than two decades, using over 200,000 lines of code and hundreds of heuristics written by domain experts. `!analyze` also provides an extensibility mechanism that both Microsoft and external developers use to build plug-ins for extending or overriding the default logic with application-specific rules. `!analyze` currently has 50+ such extensions. The extensions can be nontrivial. For example, the extension for Edge consists of over 2K LOC!

`!analyze` source code has been through many years of improvement and updates for analyzing different error codes and buckets. To understand the pain points of `!analyze`, we talked to several application owners. At present, the rules for crash report analysis are required to be written into the code. This results in huge deployment overheads; for instance, updating or deploying new rules can range anywhere between one to three months. Further, the rules often tend to be very specific, and need to be updated as the application code base evolves. For new applications, the application owner and `!analyze` developers need to work together to implement the logic of the new rules into the code base. Also, bringing up a new application is usually time consuming and requires deep domain knowledge of `!analyze` code base. With DeepAnalyze, our goal is to eliminate or significantly simplify this laborious and time consuming process with an agile and fast data-driven solution.

Table 1: Basic statistics of the dataset used in our study

# of crash stacks	$\approx 362K$
# of unique software components	$\approx 8.7K$
# of unique binaries	$\approx 16.3K$
# of unique namespaces	$\approx 38K$
# of unique methods	$\approx 85K$
# of unique blamed methods	$\approx 18K$

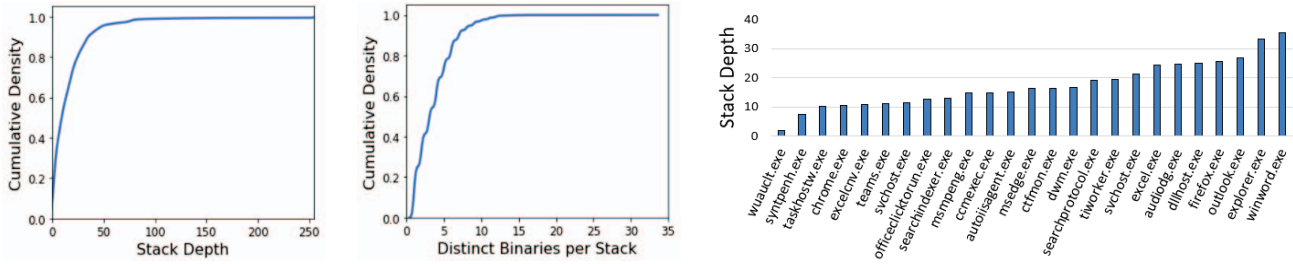
3 EMPIRICAL ANALYSIS OF CRASHES

In this section, we analyze a large collection of crashes collected by WER to understand various properties of crash stacks. Our dataset contains 362, 249 unique crash stacks, uniformly sampled from crashes collected by WER in a 1 week period. We also study properties of their problem classes and blamed frames/methods, as determined by `!analyze` with its manually written heuristics.

3.1 Crash sources

WER collects crash stacks from thousands of applications developed by both Microsoft and non-Microsoft developers. Our sample dataset contains crash stacks from $\approx 16.3K$ binaries of $\approx 8.7K$ software components[2], including popular Microsoft applications

[2]We use the term *software component* to denote various types of software systems including user-mode applications, OS or infrastructure systems, libraries, drivers, etc.



(a) Stack depth **(b) Distinct binaries per stack** **(c) Avg stack depth for top-25 software components**

Figure 4: Properties of crash stacks

such as Excel, Word, Outlook, Edge, etc. The crash stacks include $\approx 85K$ unique methods from $\approx 38K$ unique namespaces. Table 1 summaries the statistics.

Figure 3 shows distributions of top binaries and software components where crashes occur. The distribution has a long tail, with most applications contributing a small fraction of crash stacks. As shown in Figure 3 (a), there are only 10 binaries in our dataset, each of which accounts for $> 1.5\%$ (and the top binary accounts for $\approx 3\%$) of the total crashes. We also analyzed the types of top 100 software components in terms of their crash frequencies. As shown in Figure 3 (b), 75% of them are user-mode applications (e.g., Microsoft Excel) or underlying systems (e.g., Windows Desktop Window Manager (dwm)), with the remaining 25% being drivers, libraries, and services. This shows that in a large-scale error reporting system like WER, crash stacks come from many different sources. Hence, application-specific mechanisms to identify blame frames do not scale well.

Finding #1 ⇒ Crash stacks come from many different sources and hence application-specific crash localization does not scale well.

3.2 Crash stacks

Stack depth. Stack depth (i.e., number of frames in the stack) at the time of a crash indicates how deep the crash happens, in terms of nested method calls. Figure 4 (a) shows the distribution of the depths (i.e., number of frames) of all crash stacks in our dataset. Mean and median depths are 16 and 9 respectively. While majority have fewer than 10 frames, some stacks are very deep (maximum 255 frames).

Frames in a stack may contain methods from multiple binaries when one binary calls methods from another. Figure 4 (b) shows the distribution of distinct binaries appearing in a stack (average ≈ 4). Multiple binaries in a stack indicate that crashes can happen in binaries outside of the entry binary of an application.

Stack depth and software types. Stack depth varies a lot across software components and their types. For example, we find that device drivers usually have smaller stack depth (average ≈ 7) than applications (average ≈ 11) and systems (average ≈ 18). This is most likely because, compared to applications/systems, drivers are less complex in terms of the number of dependencies, and hence tend to make fewer nested method calls. Figure 4 (c) shows the average stack depths of 25 most frequent software components. The average depths differs significantly (from 1 to 35) across these software components.

Finding #2 ⇒ Stack depths significantly differ across software components and their types.

3.3 Problem classes and blamed frames

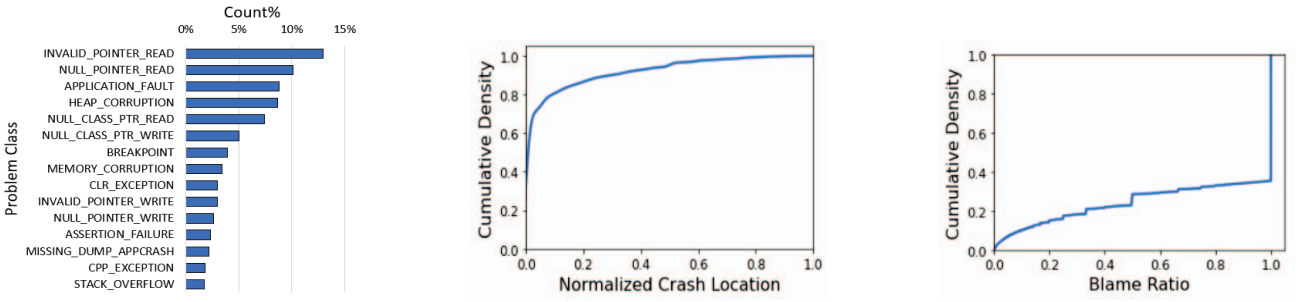
Problem classes. Figure 5a shows top 15 classes of problems that cause the crashes. We obtain problem classes from `!analyze`, which uses several heuristics to identify them from the exception context, stack traces, and other information in the crash dump. As shown, most of the key problem classes are memory related, for instance, when the application is trying to read a pointer that points to invalid memory (INVALID_POINTER_READ) or is trying to read a pointer that is null (NULL_POINTER_READ). These memory-related classes account for 61% of all crashes. Another major problem category is APPLICATION_FAULT, which is the default class when no more specific class could be identified.

Finding #3 ⇒ Most (61%) of the crashes are caused by memory-related errors.

Blamed frame location. Figure 5b shows the distribution of *normalized location* of a blamed frame in its crash stack. Locations are normalized so that the top frame and the bottom frame in a stack have locations 0 and 1 respectively. As shown, frames at the top of stacks are more likely to be blamed. More specifically, the topmost frame is indeed the blamed frame in 67% crash stacks. In the remaining 33% cases, however, blamed frame is not the top frame. An example is shown in Figure 1 (a) where the third frame is the blamed frame, and top two frames correspond to harmless logging methods. Also, in 5% cases, blamed frame is at the bottom half of the stack.

Finding #4 ⇒ Blamed frames are more likely to be located at the top of the stack. In 33% cases, however, blamed frame is below the top frame.

Context dependence. Can a method, once identified as the blamed method in a crash stack, always be blamed in other crash stacks? If yes, one could easily identify the blamed method in a new crash stack by matching its methods with a list of known blamed methods (*blame-list*). Does this simple blame-list-based approach work?



(a) Top-15 problem classes in crashes (b) Normalized crash location (c) Distribution of blame-ratio

Figure 5: Properties of problem classes and blame frames

To answer this, we compute *blame ratio* of methods. The blame ratio of a method, is the ratio of the number of crash stacks where the method is the blamed method, to the number of crash stacks containing the method. Figure 5c shows the distribution of blame ratio in our dataset. While a large fraction of methods have a blame ratio of 1, a significant fraction of methods have blame ratios less than 1. A ratio less than 1 means that a blamed method in one stack may not always be blamed in other crash stacks, and hence a blame-list-based approach would not work for these methods.

Why do some methods have blame ratios less than 1? A closer examination reveals that whether a method is indeed a blamed method, often depends on its *context* – methods that appear in frames above and below it. Let's consider the example in Figure 1 again. Here, the method `ReleaseDrawTexture` is identified as the blamed method in the first example where it appears below two relatively-harmless logging methods. On the other hand, the same method is not blamed in the second example where more crash-prone driver methods appear above it. The blamed frame's dependence on context can be explained by the fact that frames in a stack are not independent; rather, they are ordered based on the caller-callee relationship of methods in the frames. The data suggests that whether a method is blamed or not depends on the call chain through which it is invoked.

Finding #5 ⇒ Whether a method is blamed or not often depends on the context it appears in, i.e., methods that appear above and below it in the stack.

4 OUR APPROACH

In this section, we use the insights from Section 3 to design a data-driven solution to the large-scale crash localization problem.

Our goal is to utilize large-scale historical crash data, consisting of crash stacks and their labelled blamed frames, to learn models that given a new crash stack can identify its blamed frame (and method). Our first attempt was to learn a linear binary classifier (e.g., Logistic Regression) that, given various features of an individual frame, predicts the likelihood of it being the blamed frame. As we show in Section 5.1, this simple model, however, did not work well for many problem classes. This is explained by one of our findings in Section 3: the context in which a method appears in a crash stack (e.g., methods that appear before and after it) plays an important role for it to be a crash method or not. We therefore aim to build models

Natural Language Processing	**Crash Dump Analysis**
Sentence: A sequence of **words**	**Stack**: A sequence of **frames**
John lives in Seattle	f_0 f_1 f_2 f_3
Sequence Labeling	**Crash Localization**
John lives in Seattle	f_0 f_1 f_2 f_3
PER O O **LOC**	!BF !BF **BF** !BF

Figure 6: Analogy between NLP and Crash Dump Analysis

and features that can effectively capture such context. We achieve this with a novel solution that formulates the crash-localization task as a sequence labeling task, as described next.

4.1 Crash localization as sequence labeling

Sequence labeling, well explored in Natural Language Processing (NLP) [43], involves assigning a categorical label to each member of a sequence of observed values. For example, Named-Entity Recognition[30, 45, 49], a sequence labeling task, can locate and label entities in a sentence as predefined categories. It treats a sentence as a sequence of words and considers context, i.e., surrounding words, of each word to identify its category. For example, in Figure 6, the named entity *John* is identified as a *Person* and *Seattle* a *Location*.

To formulate crash localization as sequence labelling, we consider a crash stack as a sequence of frames, analogous to a sentence and its constituent words. We then perform sequence labelling with a binary category label - *BlameFrame* and *!BlameFrame*. Thus, the problem is formulated as follows: *given a crash stack (i.e., a sequence of frames), label each frame with whether it is a blame frame or not.* For example, in Figure 6, the third frame (f_2) is identified as the *BlameFrame* (BF), while the rest are labeled *!BlameFrame* (!BF). For traditional sequence labeling, one can use existing models that have been proposed previously. However, applying them to crash localization requires addressing some unique challenges.

- *What features to use?* To accurately summarize a crash stack, features need to capture both semantics and domain-specific information. So, we make use of Tf-Idf [24, 36] based features, as well as features highlighted by our empirical study, such as frame-depth, that are strongly correlated to crash locations.

Table 2: Features to represent stack frames for crash localization

Feature Group	Feature Name	Description
Semantics	`namespace`	n dimensional Tf-Idf vector of the namespace
	`method`	n dimensional Tf-Idf vector of the method
Types of Code	`is_appname_in_frame`	Does the frame contain the application's name?
	`is_first_app_frame`	Is it the 1^{st} frame with the application's name?
	`is_kernel_code`	Does the frame contain kernel code?
	`is_ntdll_code`	Does the frame contain `ntdll` code?
	`is_exception_in_frame`	Does the frame contain an exception?
Other	`norm_frame_position`	Normalized position of the frame
	`is_method_unknown`	Is the method `unknown`?
	`is_method_empty`	Is the method empty?
	`is_binary_unknown`	Is the binary `unknown`?
	`is_empty_frame`	Is the entire frame empty?

- *How to blame exactly one frame?* In traditional sequence labeling, it is possible for multiple tokens to be labeled as the same category; e.g., sentence with multiple places. In a crash stack, however, there is only one blame frame, and hence, we need appropriate models that satisfy such constraints.

In the rest of the section, we describe the design and architecture of our DeepAnalyze model that addresses these challenges.

4.2 Model Features

Guided by domain expertise and our empirical study, we engineered features shown in Table 2, for data-driven models. These features transform individual frames in the crash stack into real-valued vectors that can be used by our models. The features are generic, they apply to crashes across applications, and can be grouped into the 3 broad types briefly described below:

Semantics: These features represent the important contents of a frame such as the namespace and method name. Here, we observe that tools such as `!analyze`[16] utilize a large list of allow-lists and heuristics to localize crashes deeper in the stack. Such approaches do not consider the global *semantics* and *relevance* of the function in a frame, i.e., how a function contributes to the crash. To include these semantics, we utilize a simple Term Frequency - Inverse Document Frequency (Tf-Idf) vectorization method [24, 36]. With this approach we automatically extract a weighted list of important tokens from namespaces and methods in frames.

Types of Code: Code from applications (1*st* and 3*rd* party) are more likely to have bugs and cause crashes, when compared to to kernel code and core OS user-mode code [16]. To incorporate such information, we use features that check the presence of the application's name in the frame (i.e. the binary name). We also extract features that represent kernel code, core OS modules, and exceptions. These features can help models de-prioritize frames that are less likely to contain the root cause for crashes.

Other Information: As shown in Section 3, frames at the top of stacks are more likely to be blamed. We thus utilize the normalized frame position to model how deep in the stack the crash location can be. Also, at times frames can be incomplete or have missing symbols in scenarios such as some 3*rd* party software, and Linux OS components or standard libraries. To de-prioritize such frames, we use multiple boolean features that check for unknown and missing information in the frame.

By transforming frames using the features described in Table 2, a stack can now be visualized as a sequence of featurized frames.

4.3 DeepAnalyze Model

In the following subsections, we describe components of our multi-task model, as shown in Figure 7, in detail.

4.3.1 ***Model Overview.*** As our empirical analysis in Section 3 shows, whether a stack frame is blamed or not often depends on its context – frames above and below it. Hence, while modeling, we need to consider context flowing in both directions in the stack – top-down and bottom-up. For this, we utilize a Bi-directional LSTM (Bi-LSTM), that interprets the stack, both forwards and in reverse.

While, the BiLSTM can model sequential context flow, dependencies between frames can be widely distributed in the stack. Also, as shown in Section 3, we observe that stacks can be very long, and BiLSTMs can sometimes fail to handle long-range dependencies [56]. To overcome these challenges, we implement an Attention mechanism. It favours the model to attend to sections of the stack more likely to have the crash location.

With a Bi-LSTM and Attention layer, DeepAnalyze encodes frames and stacks into robust neural representations that can be used to localize crashes. Here, we see that unlike sequence labeling for natural language, there is a constraint where we can only label a single frame in the stack as the blame frame. To learn such structural constraints, we use a Conditional Random Fields (CRF) layer. It is a discriminative classifier that models decision boundaries between labels in a sequence.

Lastly, context for crash localization can also be external information that complements the stack. Specifically, symptoms (problem classes) associated with a crash, such as `Invalid_Pointers`, `Zero_Division`, and `Heap_Corruption`. For instance, if the problem was a `Stack_Overflow` caused by tail recursion, then the stack would contain a repeating sequence of frames. In this case, the crash can be quickly localized by attending to the repeating pattern. Based

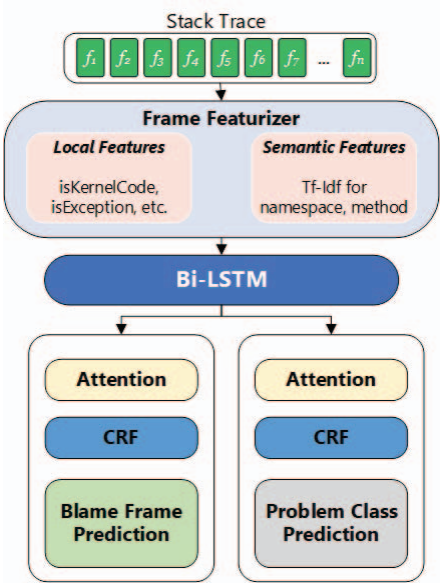


Figure 7: DeepAnalyze Multi-Task Model Architecture

on these insights, we utilize multi-task learning to perform *problem class prediction* alongside the primary task - *blame frame prediction*.

4.3.2 ***Bi-directional LSTM.*** Long Short-term Memory (LSTM) networks [21] are a type of Recurrent Neural Networks (RNNs) that have been widely used to process sequential data in a variety of tasks such as language modelling [41, 52], speech processing [46], and code comment generation [22]. It takes a sequence of inputs $(x_1, x_2, \ldots, x_n)$ as and return a sequence of vectors $(h_1, h_2, \ldots, h_n)$ that encodes information at every time step (i.e., frame level here). In our scenario, a frame f, receives context from other frames that occur on either sides. We achieve this representation using a second LSTM layer interpreting the sequence in reverse, i.e., a Bi-Directional LSTM (Bi-LSTM) [18]. Finally, each frame is represented by concatenating its left and right context, $h_f = [\overrightarrow{h_f}; \overleftarrow{h_f}]$.

4.3.3 ***Attention Mechanism.*** Attention mechanism [5, 38] has become a key component of state-of-the-art solutions to quantify distributed dependencies in sequences. It has been used for tasks like machine translation [5], sentiment classification [10], parsing [32], and even image classification [58]. Here, we implement attention mechanism at the frame level with a learnable parameter W_a, as described in Equations 1-3. It takes as input the hidden states $h = [h_1, h_2, \ldots, h_T]$ from the Bi-LSTM, and generates a weighted context vector h^* of the stack. This weighting mechanism urges the model to focus on sections of the stack that are more likely to have the crash location.

$$scores = W_a^T h \tag{1}$$

$$\alpha = softmax(scores) \tag{2}$$

$$h^* = \tanh(h\alpha^T) \tag{3}$$

4.3.4 ***Conditional Random Fields.*** The above discussed layers encode stack information into neural representations. Next, we move on to labeling the crash location. Here, we could simply predict labels independently for each frame. But we observe that this disregards some structural constraints in our problem. Specifically, unlike traditional sequence labeling, we can only label one frame as the crash location. To enforce such restrictions, we model the frame level labeling task jointly using linear chain conditional random fields (CRF) [27]. Given an input sequence X, the CRF layer computes the probability of observing an output label sequence y, i.e., $p(y|X)$:

$$s(X, y) = \sum_{i=0}^{n} A_{y_i, y_{i+1}} + \sum_{i=0}^{n} P_{i, y_i} \tag{4}$$

$$p(y|X) = \frac{e^{s(X,y)}}{\sum_{y' \in Y} e^{s(X,y')}} \tag{5}$$

Here, P is a probability matrix of shape $n \times k$ from the attention layer, where k is the number of distinct tags and n is the sequence length. A represents the matrix of scores for transitions between output labels. Finally, to extract labels, the layer predicts the output sequence with the highest probability - $y^* = \text{argmax}_{y' \in Y}\, p(y'|X)$. With this approach the model learns to include structural validity in predicted output sequences.

4.3.5 ***Multi-Task Learning.*** Multi-Task Learning (MTL) is an approach to improve generalization in models using the inductive bias in jointly learning related tasks [7]. In the context of classification and sequence labeling, MTL improves performance of individual tasks by learning multiple tasks simultaneously [49].

In our scenario, the primary task for DeepAnalyze is crash localization. As stated before, we observe that localizing crashes not only depends on the frames, but also on the class of problems that might have caused the crash. For instance, a crash caused due to an "Invalid Pointer" would generally be localized to the stack frames doing memory and IO operations. Consequently, we choose problem class prediction as a secondary task for our multi-task model. Particularly, as shown in Figure 7, we utilize a multi-head architecture to share low level features (BiLSTM layer). Then the architecture splits into 2 task specific branches - one for blame frame prediction and the other for problem class prediction.

For both tasks we use categorical cross entropy as the loss function. We first calculate loss individually for both objectives, say l_1 and l_2. We then compute and minimize a combined loss by averaging: $loss_c = (l_1 + l_2)/2$. During training, the objective we minimize is the combined $loss_c$. But, during back-propagation, we make sure to use the individual task losses (l_1 and l_2) to update weights of task-specific branches of the network (Figure 7). With such an approach, the shared layer (BiLSTM) is trained by both tasks, because both l_1 and l_2 update it via back-propagation. Whereas the task specific layers (Attention and CRF) are trained only on their respective individual loss functions.

4.4 Cross-Application Crash Localization

As stated in Section 2, hardcoding heuristics into code creates challenges with scale and generalizability for unknown future scenarios. Software constantly evolves as new applications, APIs, and programming languages are introduced and become popular. Handling crashes in such new cases usually requires a lot of time and deep domain knowledge to write custom rules and plugins for existing solutions. Here, learning a model instead, can help address the scalability and generalizability challenges with ever growing software.

But, even with supervised machine learning, for a new application, it is nontrivial to develop accurate crash localization models, as there would be minimal labeled training data. However, in crashes, we believe that there are many patterns to be learnt that are common across applications; especially the large portion of frames that represent the underlying system (see Figure 3(b)). This implies that models trained on crashes from a global set of applications (source) can be used to localize crashes for a new and disjoint set (target) — *Cross-Application Crash Localization*.

In this work, we use this to overcome the above mentioned challenges with a **Transfer Learning** and **Fine-tuning** approach. Formally, following Pan and Yang [44], transfer learning involves the concepts of a domain and a task. A domain $\mathcal{D}$ consists of 2 parts; a feature space $\mathcal{X}$ and a marginal probability distribution $P(X)$ where $X = x_1, \ldots, x_n \in \mathcal{X}$. Given a specific domain, $\mathcal{D} = \{\mathcal{X}, P(X)\}$, a task $\mathcal{T}$ has 2 components; a label space $\mathcal{Y}$ and an objective $f(\cdot)$ (i.e., $T = \{\mathcal{Y}, f(\cdot)\}$), that can be learned from training data. Given this, transfer learning is defined as:

***Transfer Learning*:** Given a source domain $\mathcal{D}_S$ and learning task $\mathcal{T}_S$, target domain $\mathcal{D}_T$ and learning task $\mathcal{T}_T$, transfer learning aims to improve the learning of the target predictive function $f_T(\cdot)$ in $\mathcal{D}_T$ using the knowledge in $\mathcal{D}_S$ and $\mathcal{T}_S$, where $\mathcal{D}_S \neq \mathcal{D}_T$, or $\mathcal{T}_S \neq \mathcal{T}_T$.

In our scenario, we observe $\mathcal{D}_S \neq \mathcal{D}_T$, where $\mathcal{D}_S$ is the global set of application crashes and the target $\mathcal{D}_T$ is a new/unseen application's crashes. Specifically, we see that the feature space ($\mathcal{X}$) of the source and target are same, while the marginal probability distributions $P(X)$ are different. This case is generally known as "Domain Adaptation" [15]. With that in mind, for cross-application crash localization, we first pre-train a DeepAnalyze model on a large dataset of crashes spanning multiple applications (Global model). This model learns general and common information about crashes. Then, for a new application scenario, we use transfer learning to adapt the weights of this global model to the application of interest with minimal labeled data. In Section 5, we test our hypothesis and extensively evaluate this approach.

5 EVALUATION

Implementation. We have implemented DeepAnalyze and all other machine learning models discussed in this work in Python 3.7.7, with Keras-2.2.4 and the tensorflow-1.15.0 backend. The semantic vectorizers are implemented using the standard tf-idf vectorizer in scikit-learn. For our Bi-LSTM CRF based models, the length of the sequence is limited to a maximum of 255, as collected by WER. Also, the hidden layer size is set to 200 cells along with a dropout of 25% to prevent overfitting by ignoring randomly selected neurons during training. Further, we use an early stopping mechanism to stop training when model performance on a validation dataset starts to degrade. Lastly, our models are trained on an Ubuntu 16.04 LTS machine, with 24-core Intel Xeon E5-2690 v3 CPU (2.60GHz), 112 GB memory and 64-bit operating system. The machine also has a Nvidia Tesla P100 GPU with 16 GB RAM.

Table 3: Evaluation of App-Specific Model Accuracy

Model	Application				Avg
	Edge	Excel	Word	Outlook	
`TopFrame`	0.64	0.77	0.70	0.62	0.68
`SecondFrame`	0.24	0.07	0.10	0.13	0.13
`MostFreqTopFrame`	0.31	0.42	0.39	0.39	0.38
Logistic Regression	0.86	0.81	0.75	0.69	0.77
BiLSTM-CRF-Attn	0.91	0.90	0.80	0.81	0.85
DeepAnalyze	**0.93**	**0.94**	**0.85**	**0.88**	**0.90**

We next consider two settings and evaluate the *accuracy* of DeepAnalyze: the fraction of test crash stacks for which DeepAnalyze correctly identifies the blame frame.

5.1 Application-Specific Evaluation

We first consider an *application-specific* setting where the training and test data come from the same applications. This makes sense when the target application has sufficient labelled training data.

Setup. We here use crash stacks from 4 popular client applications from Microsoft - *Edge*, *Excel*, *Word*, and *Outlook*. To evaluate in a realistic setup, we train and test our models at different points in time. We begin by collecting a sample of ≈ 1.2 million user mode crashes of these applications over a window of 2 weeks. For each application, we utilize data from the first 11 days for training and next 3 days for testing (11:3 ≈ 80%:20%). Also, we use a combined hash of the stack, blamed frame, and other metadata to de-duplicate our datasets, avoiding multiple evaluations of the same problem. Lastly, we establish ground truth using `!analyze` as used by WER.

We compare our multi-task model (described in Section 4.3) against multiple heuristics and machine learning baselines. In Table 3, we report the accuracy (ratio of #correctly localized crashes to #total crashes) and evaluate models on 4 different applications.

Heuristic Baselines. First, we have a `TopFrame` baseline that always picks the 1^{st} frame in the stack. This is based on the insight that a large proportion of crash locations are at the top of the stack. But, we also observed that in some cases the top frame is an exception raised by the method below it. We represent this using our `SecondFrame` baseline that always picks the 2^{nd} frame. Next, with `MostFreqTopFrame`, we introduce the use of frequent patterns. This baseline picks the frame that was most frequently blamed in the past. In case of ties or unseen frames, it favours frames higher in the stack. From Table 3, we see that these heuristic approaches perform well only on certain crashes and do not generalize well.

Linear Model. Next, we have a Logistic Regression baseline. It is linear binary classifier that, given features of an individual frame (described in Section 4.2), predicts the likelihood of it being the blame frame. Then, we pick the frame in the stack with maximum

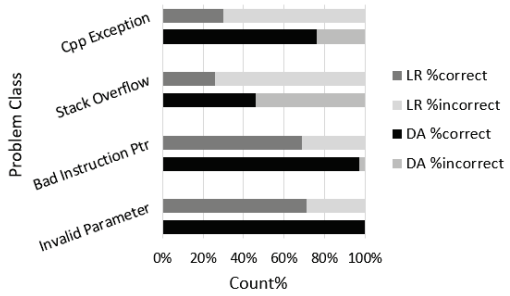


Figure 8: Logistic Regression (LR) vs DeepAnalyze (DA)

likelihood. As seen in Table 3, this simple data-driven model performs better than naive heuristics (0.77 avg accuracy). But, on further analysis, we found that this model performs poorly for specific problem classes. For instance, in Figure 8, we see that for Cpp Exception and Stack Overflow, it correctly predicts only 30% of cases. Here, we observed that such problems have diverse crashes where a method can be blamed in one crash stack but not in another. That is, the logistic regression approach lacks in capturing *context*.

Sequence Labeling. Lastly, we evaluate models that incorporate the missing context using our novel sequence labeling formulation – BiLSTM-CRF-Attn and DeepAnalyze. Here, we use BiLSTM-CRF with attention mechanism as a baseline as it is a state-of-the-art model for sequence labeling in NLP [28, 37]. Also, note that this BiLSTM-CRF-Attn model is similar to the DeepAnalyze model architecture but without multi-task learning. As shown in Table 3, it achieves an average accuracy of around 0.85. Whereas, our DeepAnalyze multi-task model (described in Section 4.3) achieves a higher average accuracy of 0.90 and beats all baselines across applications. It also reaches a maximum accuracy of 0.94 for Excel. With DeepAnalyze combining context information and complementary information using multi-task learning, we are able to outperform strong baselines such as Logistic Regression and BiLSTM-CRF-Attn. Also, though in Figure 8 we mention only some problem classes, we find that DeepAnalyze is always better than logistic regression.

Table 4: Significance of Improvements

Improvement Area	Application				Avg
	Edge	Excel	Word	Outlook	
Semantics	29%	5%	6%	10%	13%
Context	8%	15%	13%	24%	15%
Multi-Task	3%	4%	6%	9%	5%

Improvements. Table 4 shows the significance of important aspects of our approach, namely frame semantics, context dependence, and multi-task learning. We compute significance by calculating the percentage difference in accuracy (A) of pairs of models; i.e., $|A_{m1} - A_{m2}|/avg(A_{m1}, A_{m2}) \times 100\%$. To capture significance of *Semantics*, we compare Logistic Regression, that uses semantic features, with the naive `TopFrame` baseline. As shown, introducing frame features/semantics, generates considerable improvements across applications (average 13%). Next, we evaluate the value of *Context*. Here, we compare DeepAnalyze, a context-aware approach, to Logistic Regression. The 8% – 24% boosts achieved with our approach highlights the importance of context in crash localization. Also, incorporating context provides the largest gains on average (15%). Lastly, DeepAnalyze leverages both context and complementary information using multi-task learning. Thus, by comparing the multi-task DeepAnalyze model with BiLSTM-CRF-Attn, we see that multi-task learning also provides notable increases in accuracy (average 5%).

Table 5: Feature Importances

+ve feature	Imp	-ve feature	Imp
method `memory`	1.0	namespace `std`	-1.0
namespace `file`	0.73	method `error`	-0.76
method `thread`	0.49	method `exception`	-0.59

What does DeepAnalyze learn? To gain insight into what DeepAnalyze learns, we use a model weight inspection technique. It is commonly used to interpret black-box models in image processing [25, 29] and medical domains [40]. Here, we extract the weights of the 1^{st} layer of DeepAnalyze, where there is a direct interaction on the raw inputs, to generate normalized feature importances. Table 5 summarizes the top-3 positive and negative features, all of which are tf-idf semantic features. As shown, DeepAnalyze intelligently learns that methods performing memory, file, and thread operations are positive features as they work with pointers and tend to cause crashes. This is supported by our empirical analysis in Section 3 on frequent problem classes. On the other hand, DeepAnalyze also learns to negatively associate standard libraries (namespace std) and methods that raise errors/exception with crash locations, supporting our observation in Figure 3(b) that crashes are relatively less frequent in libraries.

5.2 Cross-Application Evaluation

We now evaluate DeepAnalyze in a *cross-application* setting where we have a recent/new application with minimal labeled crashes. For this, we attempt to evaluate the efficacy of our transfer learning approach for cross-application crash localization (Section 4.4). To summarize, we hypothesis that models learnt from crashes of a set of applications can localize crashes in new/unseen applications. Here, we evaluate our hypothesis on 2 target applications – *Edge* & *Excel*.

Setup. We start with the dataset used in Section 3, consisting of $\approx 362K$ crash stacks from many software components, sampled over a period of 1 week. To simulate a cross-application setting, we choose a target application and remove all its associated crashes from our dataset. Then, we train a DeepAnalyze model (Global Model) on the resultant dataset and test on crashes of the target (domain transfer). Further, we fine-tune our global model to the target application with minimal labeled crashes. Similar to the application specific evaluation in Section 5.1, for each target application, we utilize uniformly sampled data from the first 11 days for fine-tuning and next 3 days for testing. To evaluate, we compare this transfer learning approach against an application-specific DeepAnalyze model (Local Model).

Figure 9 show the results of our experiments for 2 target applications - Edge (left) and Excel (right). The `Fine-Tune-Global` (blue) line indicates the accuracy of our global model on fine-tuning (i.e., our transfer learning approach). The `From-Scratch-Local` (red)

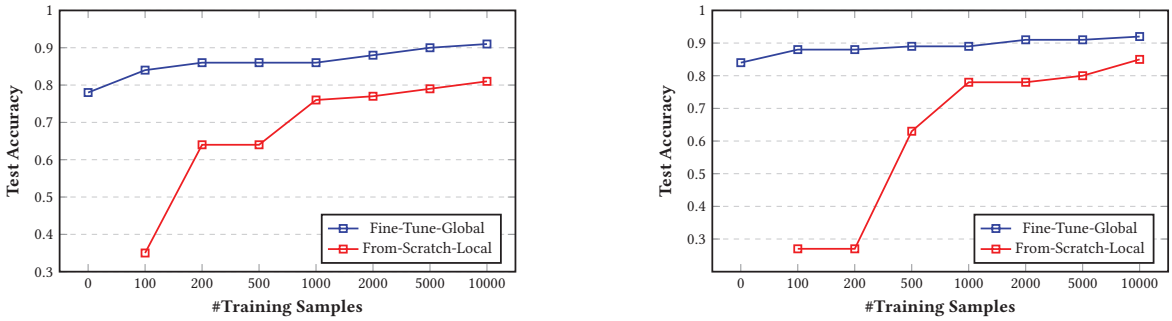


Figure 9: Fine-tuning vs Training from scratch for Edge (left) and Excel (right)

line indicates the accuracy of an application-specific model trained from scratch. The X-axis has the number of samples used for fine-tuning the global and training the local model, respectively.

Accuracy. As seen, for both Edge and Excel, our transfer learning approach significantly outperforms a local model trained from scratch, at all amounts of training data. This is mainly because a global model receives a significant head start by learning signals and patterns in crashes that are application agnostic, and hence, transferable. Specifically, we observe that our global model achieves high accuracies (0.78 for Edge; 0.84 for Excel), without observing a single application-specific crash (@ 0 training samples). This shows that DeepAnalyze can indeed learn from a global crash dataset to effectively localize crashes in new/unseen scenarios. Also, we observe that our models gradually improve with minimal labeled data. For instance, using transfer learning, we achieve ≈ 0.90 accuracy with as few as 1000-2000 samples, for both Edge and Excel. This encourages that our transfer learning approach can be used in a real-word setting, where for a new application at first we directly use a global model. But, over time, we would collect app-specific labeled data and improve our models.

Cost Savings. Furthermore, during our experiments we observed that transfer learning not only reduces training data requirements, but also training time compared to application-specific models. To evaluate, we make use of our Edge and Excel models trained on large datasets in Section 5.1. Note, these models are comparable to the fine-tuned models because they are tested on the same set of crashes. These models took on average 3 hours to train and received an average accuracy of 0.93. On the other hand, from Figure 9, comparable global models (≈ 0.91 acc with 5000-10000 samples) took on average 10 minutes to be fine-tuned. That is, we see nearly 18X reduction in training time, and thus compute cost, with a transfer learning approach. This is particularly encouraging of the usability of such an approach to quickly develop accurate models for newly deployed scenarios.

6 RELATED WORK

Crash Analysis. Debugging and triaging of crashes at scale can be expensive and intractable. Our work is most closely related to prior work on large-scale analysis of real world software crashes. The Windows Error Reporting (WER) [16] distributed system was built by Microsoft for collecting and analyzing crash traces. With DeepAnalyze, we leverage a novel machine learning based approach for crash localization using the crash traces collected by WER in the wild. Our approach not only generalizes well for several existing applications but can also extend to new applications with very limited amount of labelled data using transfer learning. Wu et al. [59] proposed CrashLocator which uses call stack information in the crash reports along with the static call graph information from the source code to predict the blame probability of each frame in the stack. In DeepAnalyze, since we are doing crash localization at scale in the wild, we don't have access to the source code. So, we only use the call stacks from the crash traces to do the crash localization. Further, we leverage recent advances in the machine learning and NLP domain for crash localization. Crash bucketization is an important part of triaging crash reports. The WER system leverages more than 500 heuristics for bucketing. Dang et al. [14] proposed ReBucket which uses crash stack similarity to assign them to appropriate buckets. Similarly, TraceSim [55] leverages TF-IDF and Levenshtein distance on crash reports for measuring similarity of crash traces for better triaging and de-duplication. Our work is complementary to these efforts since more precise crash localization can aid with crash bucketization and triaging.

Multi-task Learning. Multi-task Learning (MTL) [7, 62] is a well-studied technique in the machine learning community. MTL is used to improve the generalization and performance of ML models on a given task by jointly training on other related tasks. MTL has been utilized in several domains such as NLP [9, 13, 35, 39, 51], speech [3, 11, 26, 50, 60] and healthcare [6, 8, 20, 23, 61]. In the NLP domain, Collobert et al. [13] proposed a novel convolutional network architecture which uses MTL to jointly perform several NLP tasks such as POS tagging, named-entity extraction and measuring semantic similarity.

In the software engineering domain, MTL has been leveraged to a limited extent. Prior work has heavily focused on using MTL for building language models for source code [33, 34, 57] and software artifacts like bug reports and discussions [17, 31, 48, 49]. Wang et al. [57] propose MulCode, a MTL based approach to learn a unified representation of source code by jointly training on three tasks: author attribution, comment classification and duplicate function detection. Their evaluation show the efficacy of MTL by outperforming state of the approaches which addressed these tasks separately. To the best of our knowledge, DeepAnalyze is the first effort to use MTL in context of debugging. It leverages MTL to jointly

perform crash localization and problem class identification from crash stacks. Based on the experiments on several popular applications, MTL significantly boosts the accuracy.

7 DISCUSSION & CONCLUSION

In this paper, we proposed a novel data-driven solution to address the crash-localization problem at scale. We presented the first large-scale empirical study of 362K crashes and their blamed methods reported to WER by many Microsoft applications running in the wild. The analysis provides valuable insights on where and how the crashes happen and what methods to blame for the crashes. These insights enable us to develop DeepAnalyze, a novel multi-task sequence labeling approach for identifying blamed frames in a stack trace. We evaluate our model with real-world crashes from four popular Microsoft applications and show that DeepAnalyze, when trained with crashes from one application, can not only accurately localize crashes (with $\approx$ 90% accuracy) of the same application, but also bootstrap crash localization for other applications with zero to very little training data. This makes DeepAnalyze a practical solution to be used in the wild for a large number of applications.

As next step, we are planning to integrate DeepAnalyze with the WER service along with a feedback loop. Using the feedback provided by developers, we will train DeepAnalyze in an online learning setting. While in this work we tackle the fundamental problem of crash localization, systems like WER aid in various other tasks. For instance, they also perform crash bucketization and root cause hypothesis testing. Current solutions for these tasks, similar to crash localization, are mostly rule based which does not scale and generalize easily to new scenarios. Lastly, we will also be looking at new problems like inter-crash dump correlation when there are multiple OS running on a single device, such as in gaming consoles. Similarly, cross-platform and cross-OS crash localization in a data efficient manner is also critical. We plan to extend DeepAnalyze to solve these challenges with the overarching goal of simplifying debugging in the large.

REFERENCES

[1] 2021. About WER. https://docs.microsoft.com/en-us/windows/win32/wer/about-wer. Accessed: 2021-08-30.

[2] 2021. Using the !analyze Extension. https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/using-the--analyze-extension. Accessed: 2021-08-30.

[3] Antonios Anastasopoulos and David Chiang. 2018. Tied multitask learning for neural speech translation. *arXiv preprint arXiv:1802.06655* (2018).

[4] Apple. 2010. Diagnosing Issues Using Crash Reports and Device Logs. https://developer.apple.com/documentation/xcode/diagnosing-issues-using-crash-reports-and-device-logs.

[5] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).

[6] Jinbo Bi, Tao Xiong, Shipeng Yu, Murat Dundar, and R Bharat Rao. 2008. An improved multi-task learning approach with applications in medical diagnosis. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 117–132.

[7] Rich Caruana. 1997. Multitask learning. *Machine learning* 28, 1 (1997), 41–75.

[8] Rich Caruana, Shumeet Baluja, Tom Mitchell, et al. 1996. Using the future to" sort out" the present: Rankprop and multitask learning for medical risk evaluation. *Advances in neural information processing systems* (1996), 959–965.

[9] Soravit Changpinyo, Hexiang Hu, and Fei Sha. 2018. Multi-task learning for sequence tagging: An empirical study. *arXiv preprint arXiv:1808.04151* (2018).

[10] Peng Chen, Zhongqian Sun, Lidong Bing, and Wei Yang. 2017. Recurrent attention network on memory for aspect sentiment analysis. In *Proceedings of the 2017 conference on empirical methods in natural language processing*. 452–461.

[11] Zhuo Chen, Shinji Watanabe, Hakan Erdogan, and John R Hershey. 2015. Speech enhancement and recognition using multi-task learning of long short-term memory recurrent neural networks. In *Sixteenth Annual Conference of the International Speech Communication Association*.

[12] Chromium. 2020. Chromium CrashPad,. https://chromium.googlesource.com/crashpad/crashpad/+/refs/heads/main/README.md.

[13] Ronan Collobert and Jason Weston. 2008. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*. 160–167.

[14] 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 *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 1084–1093.

[15] Hal Daume III and Daniel Marcu. 2006. Domain adaptation for statistical classifiers. *Journal of artificial Intelligence research* 26 (2006), 101–126.

[16] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. 2009. Debugging in the (very) large: ten years of implementation and experience. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 103–116.

[17] Xi Gong, Zhenchang Xing, Xiaohong Li, Zhiyong Feng, and Zhuobing Han. 2019. Joint Prediction of Multiple Vulnerability Characteristics Through Multi-Task Learning. In *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 31–40.

[18] Alex Graves and Jürgen Schmidhuber. 2005. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural networks* 18, 5-6 (2005), 602–610.

[19] Yongfeng Gu, Jifeng Xuan, Hongyu Zhang, Lanxin Zhang, Qingna Fan, Xiaoyuan Xie, and Tieyun Qian. 2019. Does the fault reside in a stack trace? assisting crash localization by predicting crashing fault residence. *Journal of Systems and Software* 148 (2019), 88–104.

[20] Hrayr Harutyunyan, Hrant Khachatrian, David C Kale, Greg Ver Steeg, and Aram Galstyan. 2019. Multitask learning and benchmarking with clinical time series data. *Scientific data* 6, 1 (2019), 1–18.

[21] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[22] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 200–20010.

[23] Sarfaraz Hussein, Kunlin Cao, Qi Song, and Ulas Bagci. 2017. Risk stratification of lung nodules using 3D CNN-based multi-task learning. In *International conference on information processing in medical imaging*. Springer, 249–260.

[24] Karen Sparck Jones. 1972. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation* (1972).

[25] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. 2014. Large-scale video classification with convolutional neural networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*. 1725–1732.

[26] Suyoun Kim, Takaaki Hori, and Shinji Watanabe. 2017. Joint CTC-attention based end-to-end speech recognition using multi-task learning. In *2017 IEEE international conference on acoustics, speech and signal processing (ICASSP)*. IEEE, 4835–4839.

[27] John Lafferty, Andrew McCallum, and Fernando CN Pereira. 2001. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. (2001).

[28] Guillaume Lample, Miguel Ballesteros, Sandeep Subramanian, Kazuya Kawakami, and Chris Dyer. 2016. Neural architectures for named entity recognition. *arXiv preprint arXiv:1603.01360* (2016).

[29] Honglak Lee, Roger Grosse, Rajesh Ranganath, and Andrew Y Ng. 2009. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *Proceedings of the 26th annual international conference on machine learning*. 609–616.

[30] Jing Li, Aixin Sun, Jianglei Han, and Chenliang Li. 2020. A survey on deep learning for named entity recognition. *IEEE Transactions on Knowledge and Data Engineering* (2020).

[31] Mingyang Li, Lin Shi, Ye Yang, and Qing Wang. 2020. A deep multitask learning approach for requirements discovery and annotation from open forum. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 336–348.

[32] Qi Li, Tianshi Li, and Baobao Chang. 2016. Discourse parsing with attention-based hierarchical neural networks. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. 362–371.

[33] Fang Liu, Ge Li, Bolin Wei, Xin Xia, Zhiyi Fu, and Zhi Jin. 2020. A self-attentional neural architecture for code completion with multi-task learning. In *Proceedings of the 28th International Conference on Program Comprehension*. 37–47.

[34] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. 2020. Multi-task learning based pre-trained language model for code completion. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 473–485.

[35] Pengfei Liu, Xipeng Qiu, and Xuanjing Huang. 2016. Recurrent neural network for text classification with multi-task learning. *arXiv preprint arXiv:1605.05101* (2016).

[36] Hans Peter Luhn. 1957. A statistical approach to mechanized encoding and searching of literary information. *IBM Journal of research and development* 1, 4 (1957), 309–317.

[37] Ling Luo, Zhihao Yang, Pei Yang, Yin Zhang, Lei Wang, Hongfei Lin, and Jian Wang. 2018. An attention-based BiLSTM-CRF approach to document-level chemical named entity recognition. *Bioinformatics* 34, 8 (2018), 1381–1388.

[38] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. 2015. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025* (2015).

[39] Bryan McCann, Nitish Shirish Keskar, Caiming Xiong, and Richard Socher. 2018. The natural language decathlon: Multitask learning as question answering. *arXiv preprint arXiv:1806.08730* (2018).

[40] Saaed Mehrabi, Sunghwan Sohn, Dingheng Li, Joshua J Pankratz, Terry Therneau, Jennifer L St Sauver, Hongfang Liu, and Mathew Palakal. 2015. Temporal pattern and association discovery of diagnosis codes using deep learning. In *2015 International Conference on Healthcare Informatics*. IEEE, 408–416.

[41] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. In *Eleventh annual conference of the international speech communication association*.

[42] Mozilla. 2012. Mozila Crash Reports,. http://crash-stats.mozilla.com/.

[43] Nam Nguyen and Yunsong Guo. 2007. Comparisons of sequence labeling algorithms and extensions. In *Proceedings of the 24th international conference on Machine learning*. 681–688.

[44] Sinno Jialin Pan and Qiang Yang. 2009. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering* 22, 10 (2009), 1345–1359.

[45] Adwait Ratnaparkhi. 1996. A maximum entropy model for part-of-speech tagging. In *Conference on empirical methods in natural language processing*.

[46] Hasim Sak, Andrew W Senior, and Françoise Beaufays. 2014. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. (2014).

[47] Adrian Schroter, Adrian Schröter, Nicolas Bettenburg, and Rahul Premraj. 2010. Do stack traces help developers fix bugs?. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, 118–121.

[48] Manish Shetty, Chetan Bansal, Sumit Kumar, Nikitha Rao, and Nachiappan Nagappan. 2021. SoftNER: Mining Knowledge Graphs From Cloud Incidents. arXiv:2101.05961 [cs.SE]

[49] Manish Shetty, Chetan Bansal, Sumit Kumar, Nikitha Rao, Nachiappan Nagappan, and Thomas Zimmermann. 2021. Neural knowledge extraction from cloud service incidents. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 218–227.

[50] Yusuke Shinohara. 2016. Adversarial Multi-Task Learning of Deep Neural Networks for Robust Speech Recognition.. In *Interspeech*. San Francisco, CA, USA, 2369–2372.

[51] Sandeep Subramanian, Adam Trischler, Yoshua Bengio, and Christopher J Pal. 2018. Learning general purpose distributed sentence representations via large scale multi-task learning. *arXiv preprint arXiv:1804.00079* (2018).

[52] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. 2012. LSTM neural networks for language modeling. In *Thirteenth annual conference of the international speech communication association*.

[53] Ubuntu. 2008. Apport Crash Duplicates. https://wiki.ubuntu.com/ApportCrashDuplicates.

[54] Rijnard van Tonder, John Kotheimer, and Claire Le Goues. 2018. Semantic crash bucketing. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 612–622.

[55] Roman Vasiliev, Dmitrij Koznov, George Chernishev, Aleksandr Khvorov, Dmitry Luciv, and Nikita Povarov. 2020. TraceSim: a method for calculating stack trace similarity. In *Proceedings of the 4th ACM SIGSOFT International Workshop on Machine-Learning Techniques for Software-Quality Evaluation*. 25–30.

[56] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[57] Deze Wang, Yue Yu, Shanshan Li, Wei Dong, Ji Wang, and Liao Qing. 2021. MulCode: A Multi-task Learning Approach for Source Code Understanding. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 48–59.

[58] Fei Wang, Mengqing Jiang, Chen Qian, Shuo Yang, Cheng Li, Honggang Zhang, Xiaogang Wang, and Xiaoou Tang. 2017. Residual attention network for image classification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 3156–3164.

[59] Rongxin Wu, Hongyu Zhang, Shing-Chi Cheung, and Sunghun Kim. 2014. Crashlocator: Locating crashing faults based on crash stacks. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 204–214.

[60] Zhizheng Wu, Cassia Valentini-Botinhao, Oliver Watts, and Simon King. 2015. Deep neural networks employing multi-task learning and stacked bottleneck features for speech synthesis. In *2015 IEEE international conference on acoustics, speech and signal processing (ICASSP)*. IEEE, IEEE, Brisbane, 4460–4464.

[61] Daoqiang Zhang, Dinggang Shen, Alzheimer's Disease Neuroimaging Initiative, et al. 2012. Multi-modal multi-task learning for joint prediction of multiple regression and classification variables in Alzheimer's disease. *NeuroImage* 59, 2 (2012), 895–907.

[62] Yu Zhang and Qiang Yang. 2021. A survey on multi-task learning. *IEEE Transactions on Knowledge and Data Engineering* (2021).