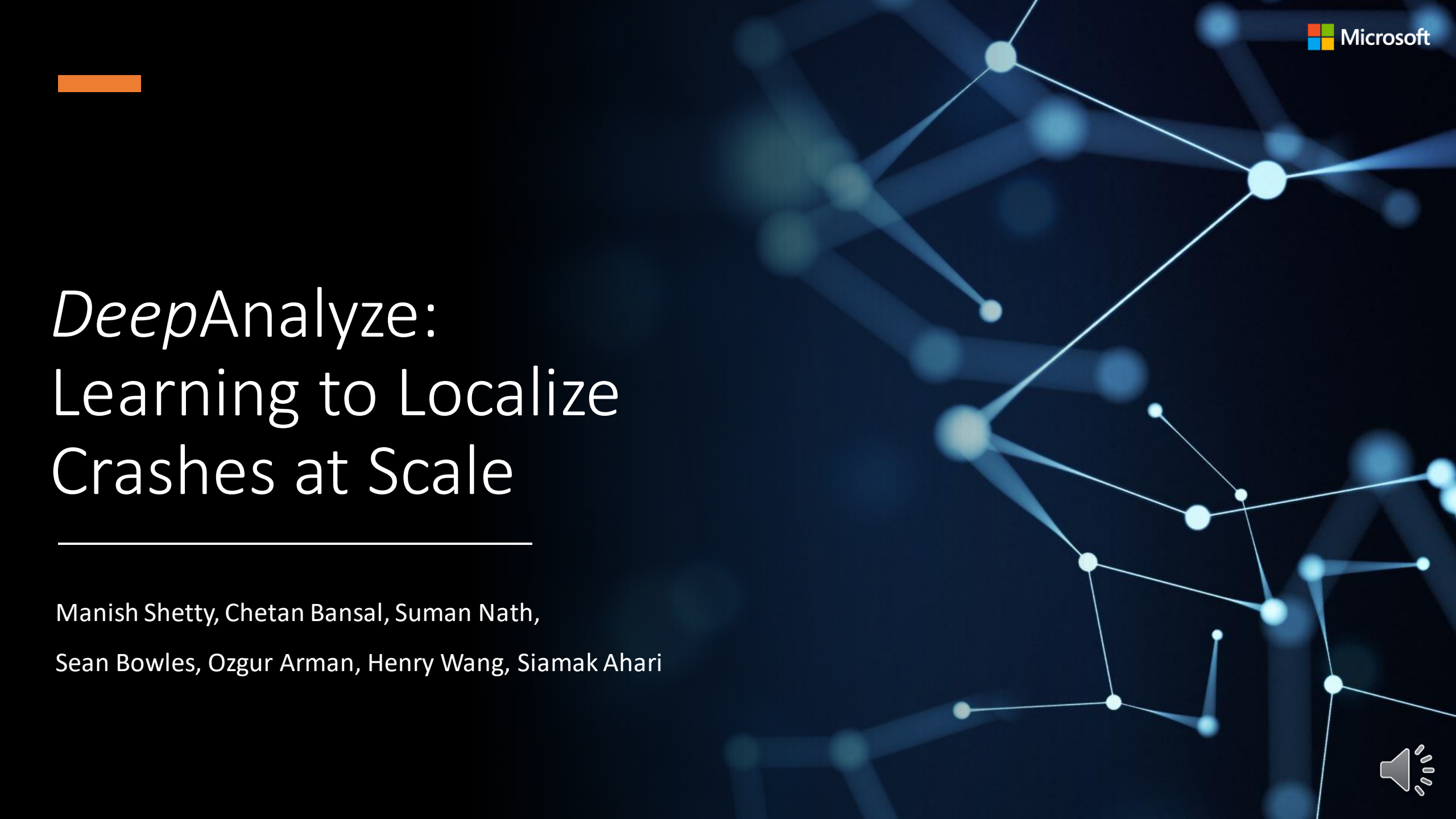
*Deep*Analyze: Learning to Localize Crashes at Scale

Manish Shetty, Chetan Bansal, Suman Nath,

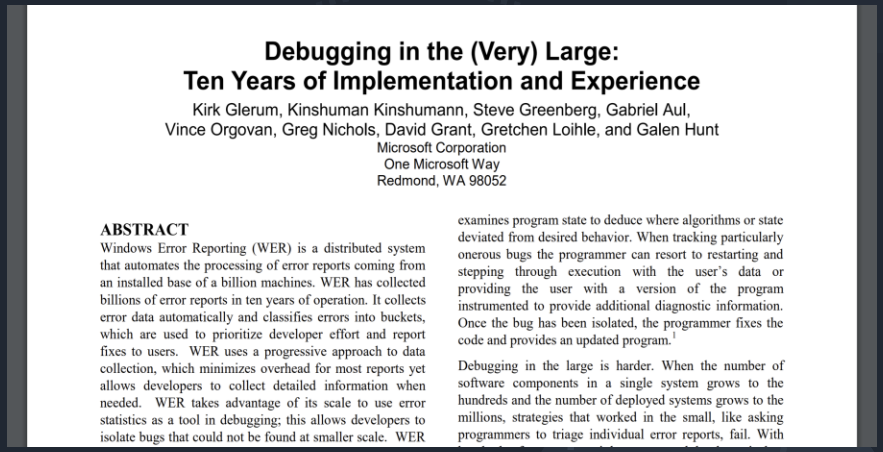
Sean Bowles, Ozgur Arman, Henry Wang, Siamak Ahari



!Analyze

A debugger extension built and maintained over 20+ years for automated analysis of crash dumps.

- Deployed as part of Windows Error Reporting (WER) Service.
- Does crash bucketization and localization.
- O(100K) lines of code and 50+ custom plugins.
- Analyzes millions of crashes per day.
- Significant usage by 1st party and 3rd party developers.



Debugging in the (Very) Large:
Ten Years of Implementation and Experience

Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul,
Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

ABSTRACT

Windows Error Reporting (WER) is a distributed system that automates the processing of error reports coming from an installed base of a billion machines. WER has collected billions of error reports in ten years of operation. It collects error data automatically and classifies errors into buckets, which are used to prioritize developer effort and report fixes to users. WER uses a progressive approach to data collection, which minimizes overhead for most reports yet allows developers to collect detailed information when needed. WER takes advantage of its scale to use error statistics as a tool in debugging; this allows developers to isolate bugs that could not be found at smaller scale. WER examines program state to deduce where algorithms or state deviated from desired behavior. When tracking particularly onerous bugs the programmer can resort to restarting and stepping through execution with the user's data or providing the user with a version of the program instrumented to provide additional diagnostic information. Once the bug has been isolated, the programmer fixes the code and provides an updated program.[1]

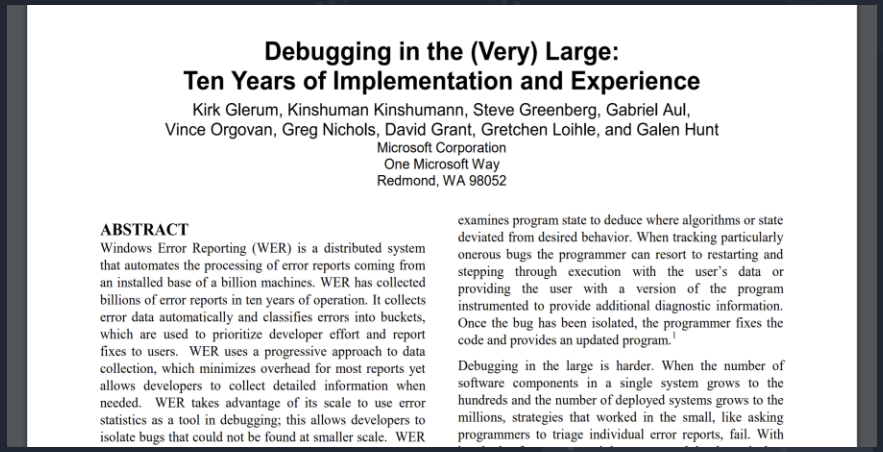
Debugging in the large is harder. When the number of software components in a single system grows to the hundreds and the number of deployed systems grows to the millions, strategies that worked in the small, like asking programmers to triage individual error reports, fail. With

SOSP **2009**

!Analyze Challenges

Despite the success, !Analyze has several limitations which needs to be addressed to maximize coverage and usability.

- Monolithic code written over two decades.
- Relies on 100(s) of heuristics encoded by domain experts.
- Code changes can take one to two months to deploy.
- New applications require custom plugins.
- Limited support for analyzing Linux and Mac OS crashes.



Debugging in the (Very) Large:
Ten Years of Implementation and Experience

Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul,
Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

ABSTRACT

Windows Error Reporting (WER) is a distributed system that automates the processing of error reports coming from an installed base of a billion machines. WER has collected billions of error reports in ten years of operation. It collects error data automatically and classifies errors into buckets, which are used to prioritize developer effort and report fixes to users. WER uses a progressive approach to data collection, which minimizes overhead for most reports yet allows developers to collect detailed information when needed. WER takes advantage of its scale to use error statistics as a tool in debugging; this allows developers to isolate bugs that could not be found at smaller scale. WER examines program state to deduce where algorithms or state deviated from desired behavior. When tracking particularly onerous bugs the programmer can resort to restarting and stepping through execution with the user's data or providing the user with a version of the program instrumented to provide additional diagnostic information. Once the bug has been isolated, the programmer fixes the code and provides an updated program.[1]

Debugging in the large is harder. When the number of software components in a single system grows to the hundreds and the number of deployed systems grows to the millions, strategies that worked in the small, like asking programmers to triage individual error reports, fail. With

SOSP **2009**

Can we augment !Analyze with data-driven approaches?

DeepAnalyze - Overview

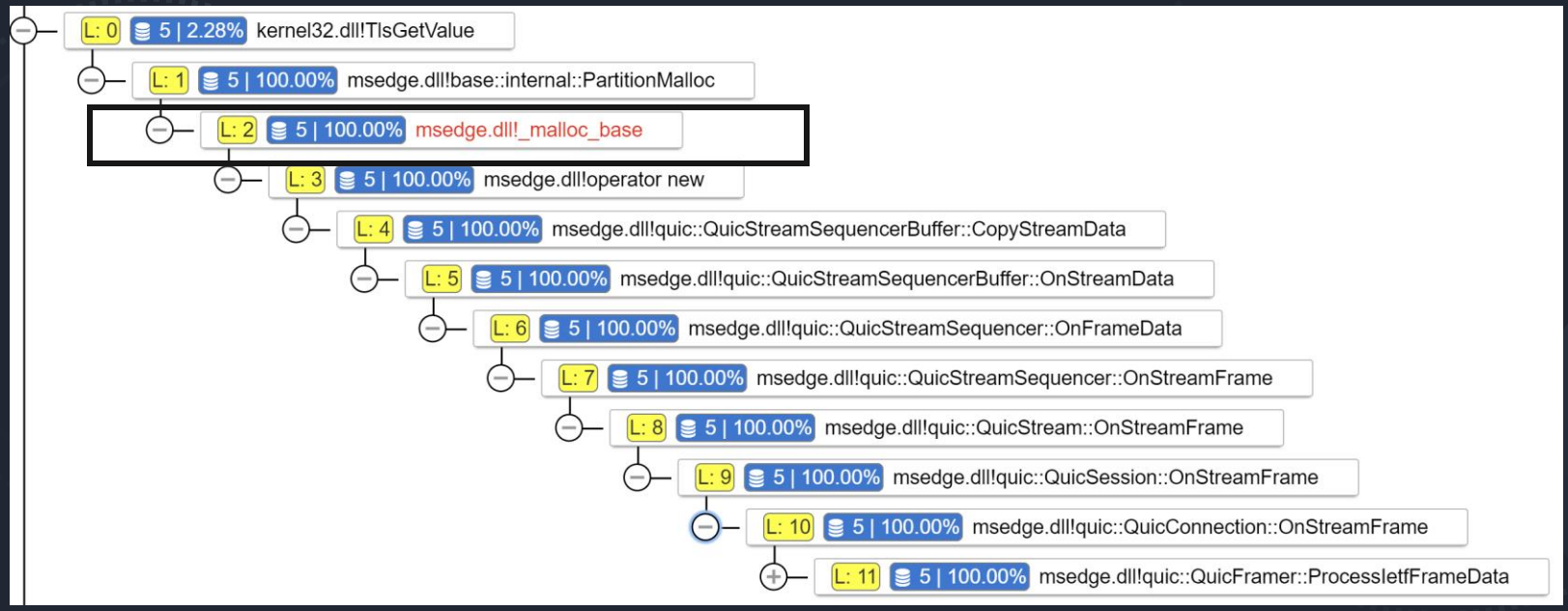
Leverage the recent advances in Deep Learning and NLP to automate crash dump analysis.

Goals

- Bootstrap using data from !analyze for ML training and treat it as the source of truth.
- Move away from manually curated rules and heuristics, reduce deployment time.
- Enable crash dump analysis for new binaries without writing custom rules and plugins.
- Build global models which can be used for analyzing Linux crashes.

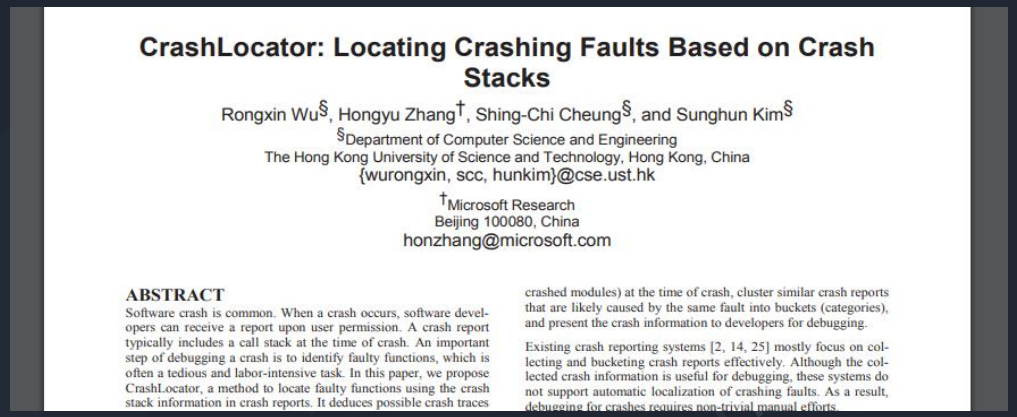
DeepAnalyze - Overview

Scenario: ML based prediction of blame frame to help developers localize the root cause.



Blame Frame Analysis

Related Work



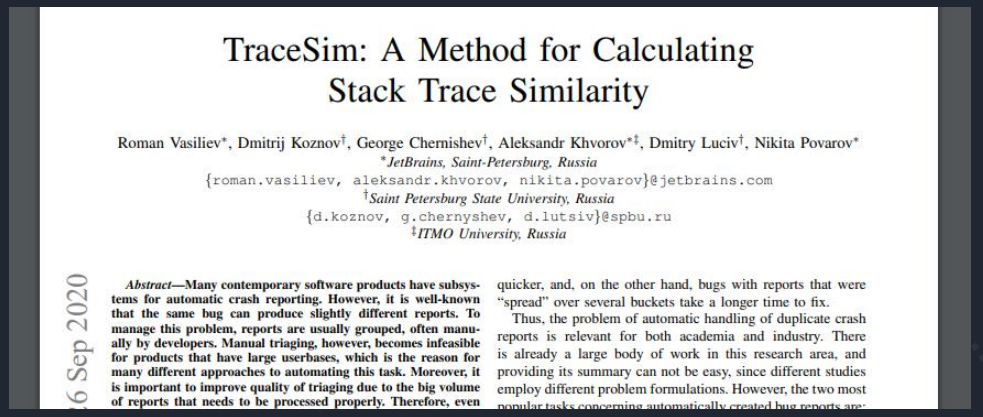
CrashLocator: Locating Crashing Faults Based on Crash Stacks

Rongxin Wu[§], Hongyu Zhang[†], Shing-Chi Cheung[§], and Sunghun Kim[§]
[§]Department of Computer Science and Engineering
The Hong Kong University of Science and Technology, Hong Kong, China
{wurongxin, scc, hunkim}@cse.ust.hk
[†]Microsoft Research
Beijing 100080, China
honzhang@microsoft.com

ABSTRACT
Software crash is common. When a crash occurs, software developers can receive a report upon user permission. A crash report typically includes a call stack at the time of crash. An important step of debugging a crash is to identify faulty functions, which is often a tedious and labor-intensive task. In this paper, we propose CrashLocator, a method to locate faulty functions using the crash stack information in crash reports. It deduces possible crash traces crashed modules) at the time of crash, cluster similar crash reports that are likely caused by the same fault into buckets (categories), and present the crash information to developers for debugging.

Existing crash reporting systems [2, 14, 25] mostly focus on collecting and bucketing crash reports effectively. Although the collected crash information is useful for debugging, these systems do not support automatic localization of crashing faults. As a result, debugging for crashes requires non-trivial manual efforts.

ISSTA ‘14



TraceSim: A Method for Calculating Stack Trace Similarity

Roman Vasiliev*, Dmitrij Koznov†, George Chernishev†, Aleksandr Khvorov*‡, Dmitry Luciv†, Nikita Povarov*
*JetBrains, Saint-Petersburg, Russia
{roman.vasiliev, aleksandr.khvorov, nikita.povarov}@jetbrains.com
†Saint Petersburg State University, Russia
{d.koznov, g.chernyshev, d.lutsiv}@spbu.ru
‡ITMO University, Russia

***Abstract*—Many contemporary software products have subsystems for automatic crash reporting. However, it is well-known that the same bug can produce slightly different reports. To manage this problem, reports are usually grouped, often manually by developers. Manual triaging, however, becomes infeasible for products that have large userbases, which is the reason for many different approaches to automating this task. Moreover, it is important to improve quality of triaging due to the big volume of reports that needs to be processed properly. Therefore, even** quicker, and, on the other hand, bugs with reports that were "spread" over several buckets take a longer time to fix.

Thus, the problem of automatic handling of duplicate crash reports is relevant for both academia and industry. There is already a large body of work in this research area, and providing its summary can not be easy, since different studies employ different problem formulations. However, the two most popular tasks concerning automatically created bug reports are:

FSE ‘20

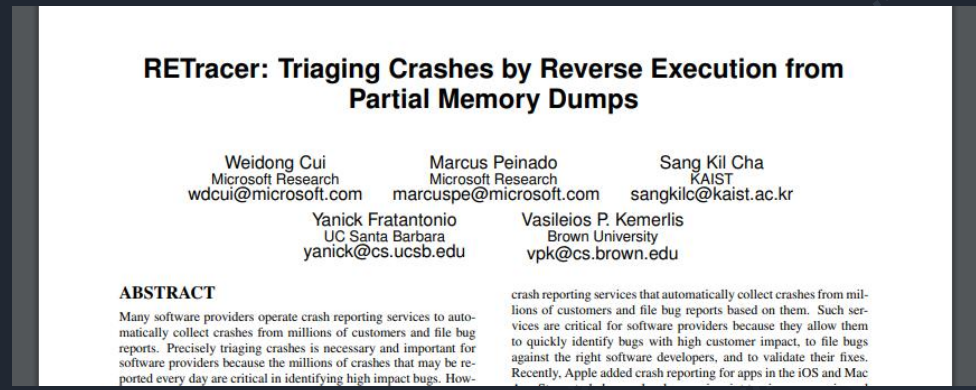
Contents lists available at ScienceDirect
The Journal of Systems and Software
journal homepage: www.elsevier.com/locate/jss

Does the fault reside in a stack trace? Assisting crash localization by predicting crashing fault residence

Yongfeng Gu[a], Jifeng Xuan[a,*], Hongyu Zhang[b], Lanxin Zhang[a], Qingna Fan[c], Xiaoyuan Xie[a], Tieyun Qian[a]

[a] School of Computer Science, Wuhan University, Wuhan 430072, China
[b] School of Electrical Engineering and Computer Science, The University of Newcastle, Callaghan NSW2308, Australia
[c] Ruanmou Edu, Wuhan 430079, China

JSS ‘19



RETracer: Triaging Crashes by Reverse Execution from Partial Memory Dumps

Weidong Cui
Microsoft Research
wdcui@microsoft.com

Marcus Peinado
Microsoft Research
marcuspe@microsoft.com

Sang Kil Cha
KAIST
sangkilc@kaist.ac.kr

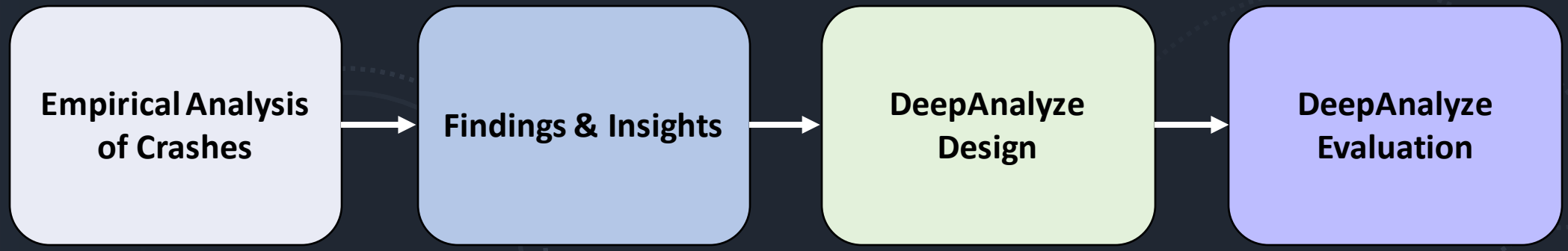
Yanick Fratantonio
UC Santa Barbara
yanick@cs.ucsb.edu

Vasileios P. Kemerlis
Brown University
vpk@cs.brown.edu

ABSTRACT
Many software providers operate crash reporting services to automatically collect crashes from millions of customers and file bug reports. Precisely triaging crashes is necessary and important for software providers because the millions of crashes that may be reported every day are critical in identifying high impact bugs. How- crash reporting services that automatically collect crashes from millions of customers and file bug reports based on them. Such services are critical for software providers because they allow them to quickly identify bugs with high customer impact, to file bugs against the right software developers, and to validate their fixes. Recently, Apple added crash reporting for apps in the iOS and Mac

ICSE ‘16

Our Approach



Empirical Analysis of Crashes

- Analyzed 362K crashes from diverse sources to understand various properties of crash stacks.
- Used insights from analysis to design and develop DeepAnalyze.

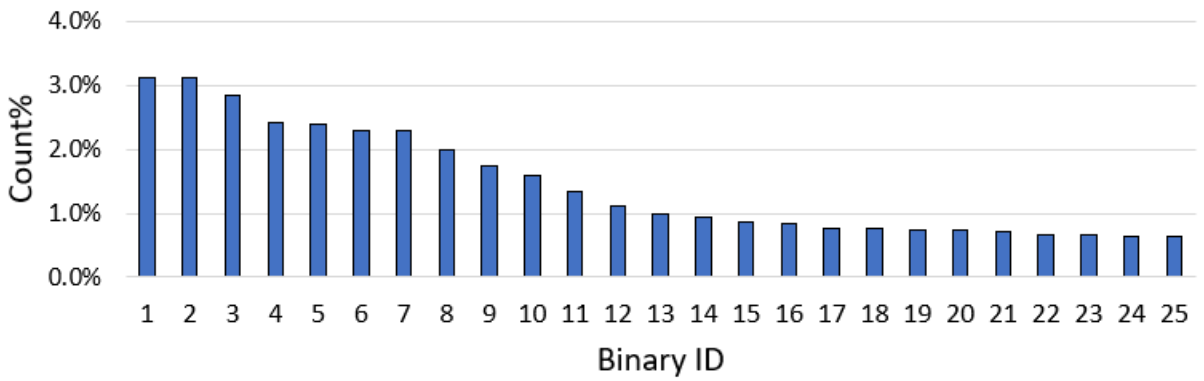
# of crash stacks	$\approx 362K$
# of unique software components	$\approx 8.7K$
# of unique binaries	$\approx 16.3K$
# of unique namespaces	$\approx 38K$
# of unique methods	$\approx 85K$
# of unique blamed methods	$\approx 18K$

Basic statistics of the study dataset

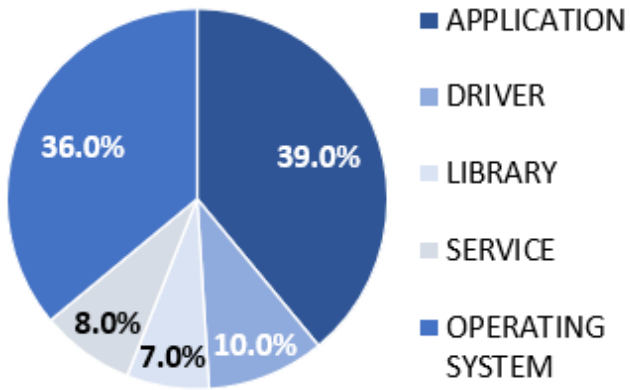
Empirical Analysis of Crashes

Finding #1

Crashes come from many different sources. Hence, application-specific heuristics for crash localization does not scale well.



Top-25 crashing binaries



Types of top-100 crashing software components

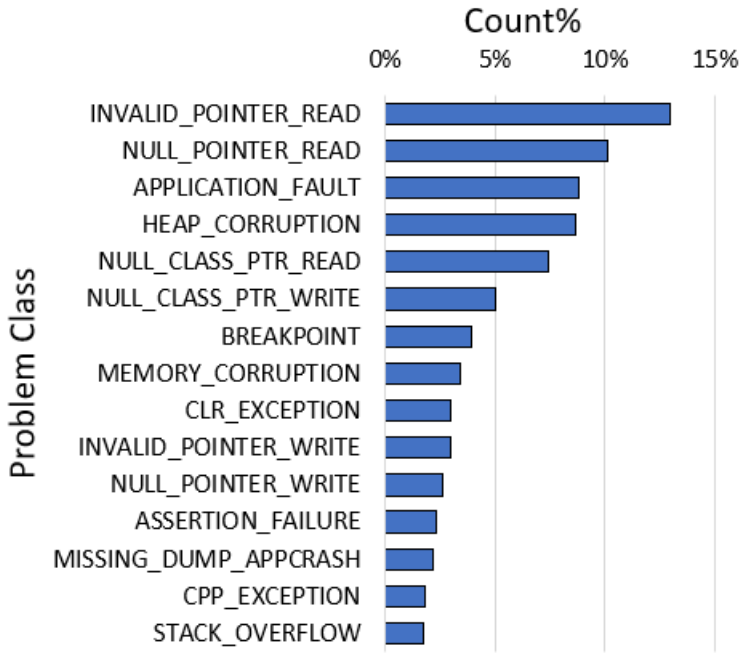
Empirical Analysis of Crashes

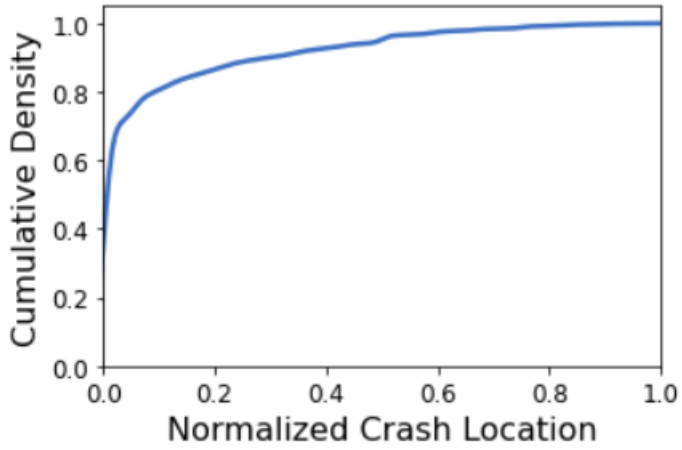
Finding #2

Most (**61**%) of the crashes are caused by memory-related errors.

Finding #3

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





Empirical Analysis of Crashes

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.

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

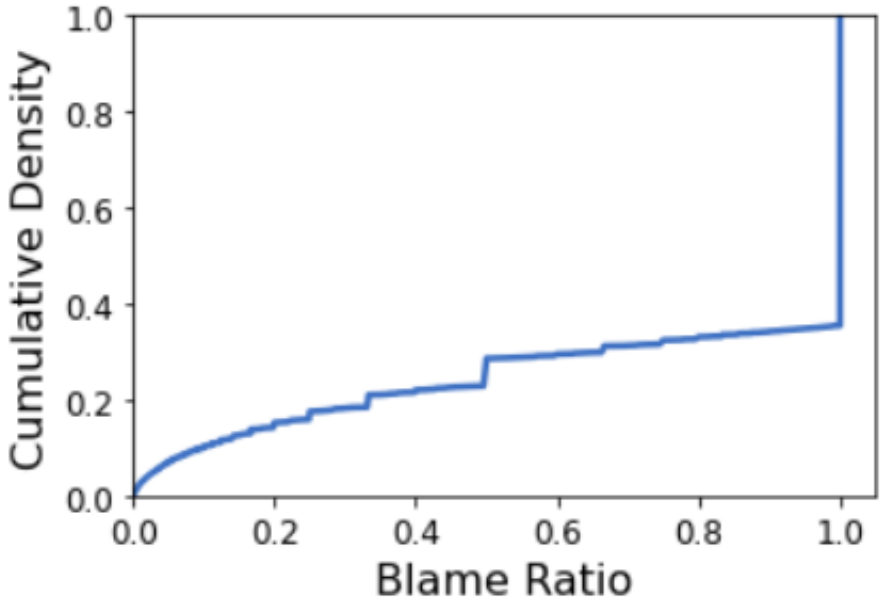
0	igd10iumd64.dll!OpenAdapter10_2
1	d3d11.dll!NDXGI::CDevice::RotateResourceIdentities
2	dxgi.dll!CDXGISwapChain::PresentImplCore
3	dxgi.dll!CDXGISwapChain::PresentImpl
4	dxgi.dll!CDXGISwapChain::[IDXGISwapChain4]::Present1
5	msedge.dll!gl::DirectCompositionChildSurfaceWin::ReleaseDrawTexture
6	msedge.dll!gl::DirectCompositionChildSurfaceWin::SwapBuffers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

(b) Crash stack 2

Empirical Analysis of Crashes

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.



$$BlameRatio(method) = \frac{\#crash\ stacks\ where\ the\ method\ is\ blamed}{\#crash\ stacks\ containing\ the\ method}$$

DeepAnalyze

- We have built several heuristics-based baseline models for blame frame prediction.
- We have built a Multi-Task learning based model for **blame frame identification.**
- Model utilizes both local and semantics features.
  - Local features: Relative Frame position, Is App Frame, Is Kernel Code, etc.
  - Semantic features: TF-IDF representation of the Namespace and the Method in the Frame.
- Model incorporates global view of a stack (i.e., context) while making predictions.

Formulation

Natural Language Processing

Sentence: A sequence of words

John	lives	in	Seattle

Sequence Labeling

John	lives	in	Seattle
PER	O	O	LOC

Crash Dump Analysis

Stack Trace: A sequence of frames

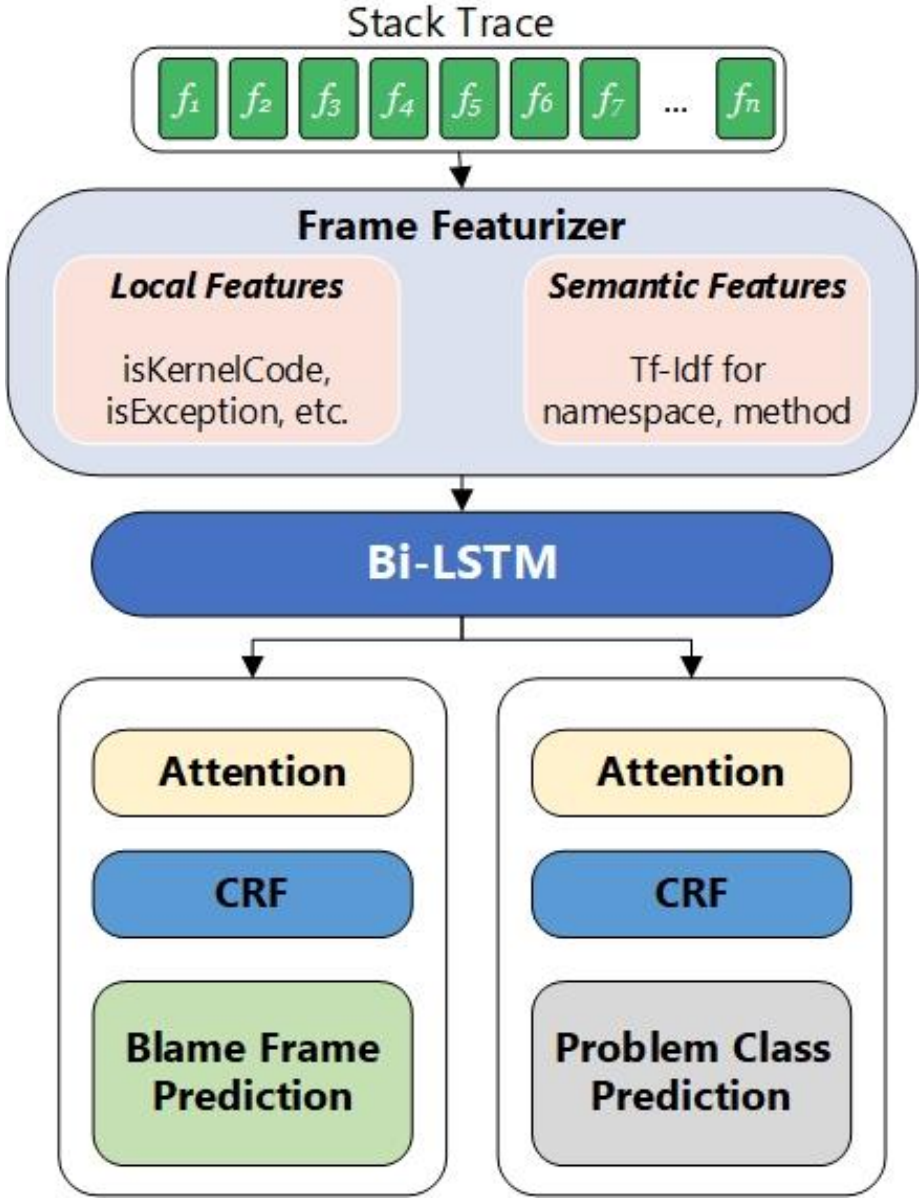
Frame_0	Frame_1	Frame_2	Frame_3

Blame Frame Prediction

Frame_0	Frame_1	Frame_2	Frame_3
notBlame	*notBlame*	*isBlame*	*notBlame*

Multi-Task Model Architecture

- Combines sequence labeling with Multi-Task learning.
- Enables joint learning from complementary tasks for better generalization.



Model Evaluation

- Baseline models using x64 User Mode crash dumps from Edge, Excel, Word, and Outlook.
- DeepAnalyze model which leverages Multi-Task sequence learning outperforms all others.
- DeepAnalyze has an average accuracy of 0.90.

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**

Model Comparison with Baselines

Model Evaluation

- Baseline models using x64 User Mode crash dumps from Edge, Excel, Word, and Outlook.
- DeepAnalyze model which leverages Multi-Task sequence learning outperforms all others.
- DeepAnalyze has an average accuracy of 0.90.

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**

Model Comparison with Baselines

Model Evaluation

- Baseline models using x64 User Mode crash dumps from Edge, Excel, Word, and Outlook.
- DeepAnalyze model which leverages Multi-Task sequence learning outperforms all others.
- DeepAnalyze has an average accuracy of 0.90.

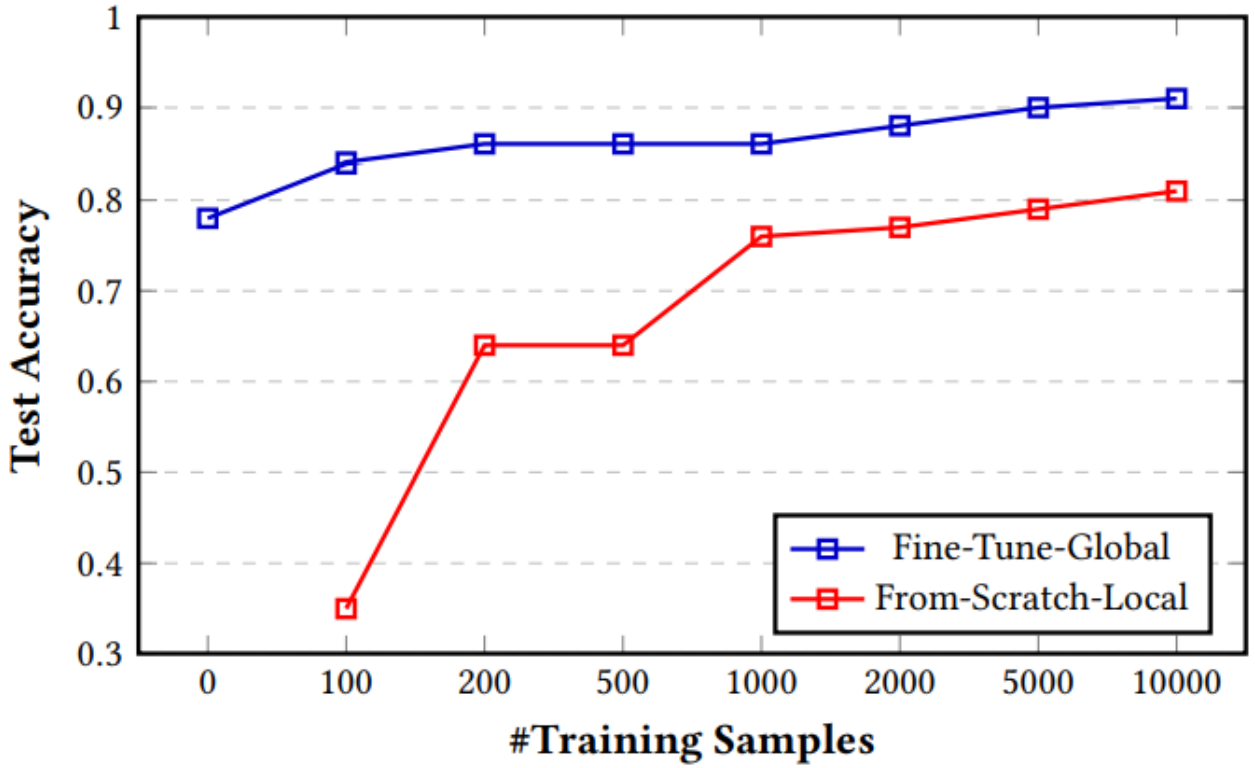
Model	Application				Avg
	Edge	Excel	Word	Outlook	
TopFrame	0.64	0.77	0.70	0.62	0.68
SecondFrame	0.24	0.07	0.10	0.13	0.13
MostFreqTopFrame	0.31	0.42	0.39	0.39	0.38
Logistic Regression	0.86	0.81	0.75	0.69	0.77
BiLSTM-CRF-Attn	0.91	0.90	0.80	0.81	0.85
DeepAnalyze	**0.93**	**0.94**	**0.85**	**0.88**	**0.90**

Model Comparison with Baselines

How do we scale to new applications and platforms?

Cross-App Crash Localization

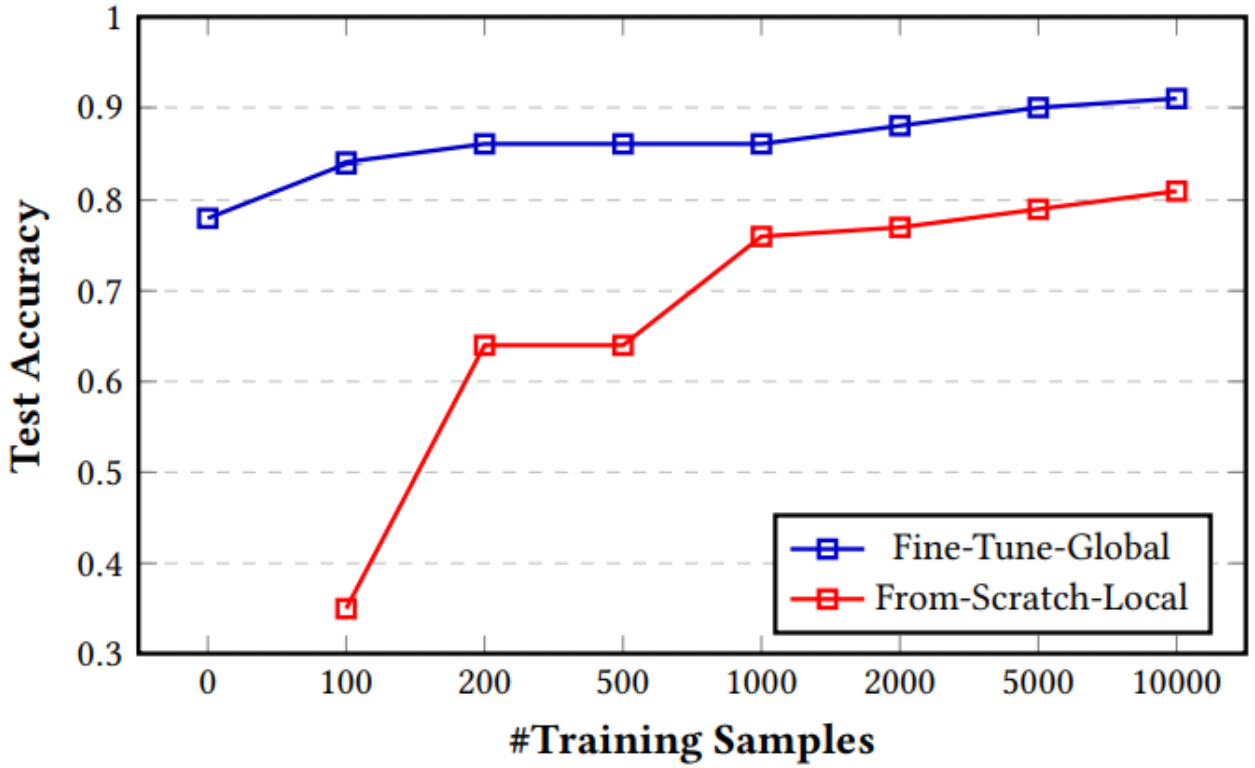
- Trained DeepAnalyze on a global dataset that does not include any Edge crashes.
- Global models can be fine-tuned to app-specific crashes with very minimal labeled data.
- Reaches ≈**0.90** accuracy with just 2000 samples.
- Near-zero training cost as opposed to training app specific models.



Fine-tuning vs Training from scratch

Cross-App Crash Localization

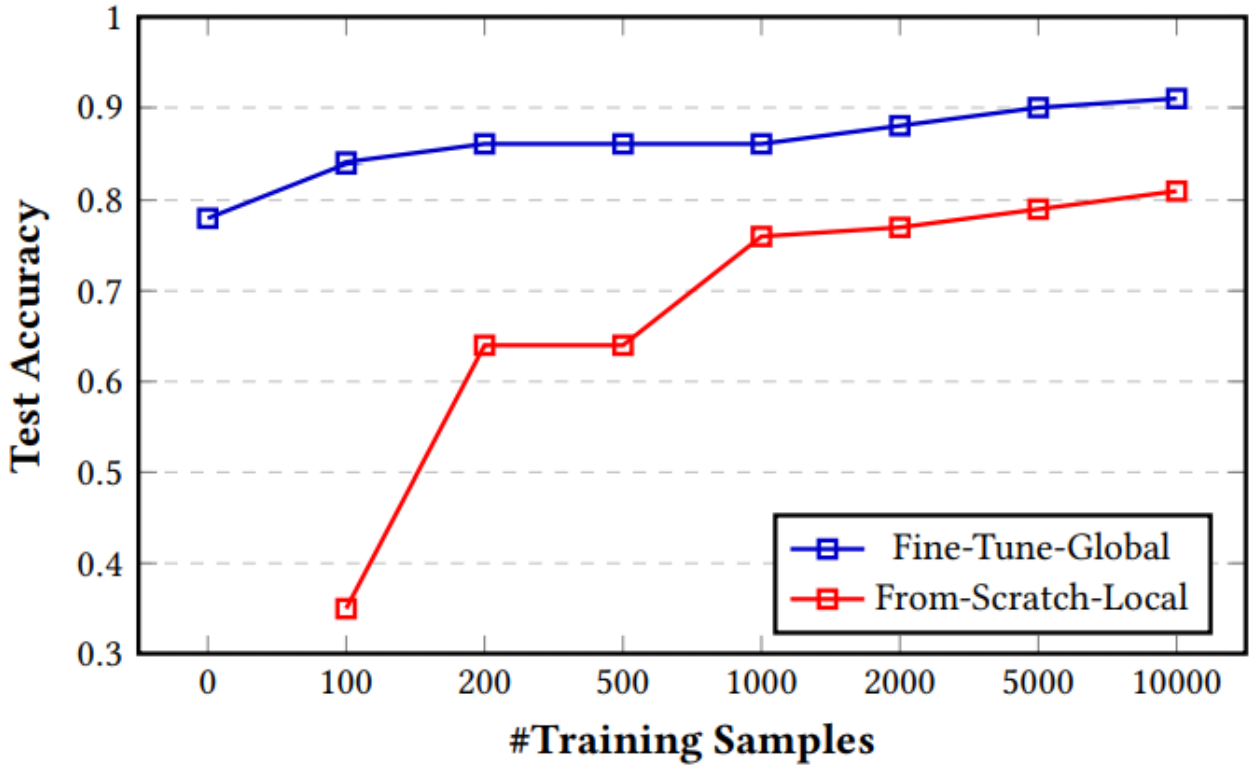
- Trained DeepAnalyze on a global dataset that does not include any Edge crashes.
- Global models can be fine-tuned to app-specific crashes with very minimal labeled data.
- Reaches ≈**0.90** accuracy with just 2000 samples.
- Near-zero training cost as opposed to training app specific models.



Fine-tuning vs Training from scratch

Cross-App Crash Localization

- Trained DeepAnalyze on a global dataset that does not include any Edge crashes.
- Global models can be fine-tuned to app-specific crashes with very minimal labeled data.
- Reaches ≈**0.90** accuracy with just 2000 samples.
- Near-zero training cost as opposed to training app specific models.



Fine-tuning vs Training from scratch

Summary

- Developed DeepAnalyze using recent advances in ML and NLP for crash localization.
- Proposed a multi-task learning based model for blame frame prediction.
- Showed an effective transfer learning and fine-tuning approach for cross-application crash localization.
- Expand to cross-platform crash localization with Linux crashes.
- Extending models to other tasks such as fault localization and bucketization at thread level.

Questions?