Scaffle: Bug Localization on Millions of Files

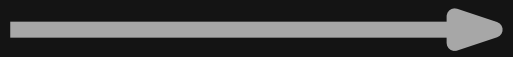
Michael Pradel[1], **Vijayaraghavan Murali**[2], **Rebecca Qian**[2], **Mateusz Machalica**[2], **Erik Meijer**[2], **Satish Chandra**[2]

[1] University of Stuttgart, [2] Facebook

ISSTA 2020

Motivation

Thousands of **field crashes** every day

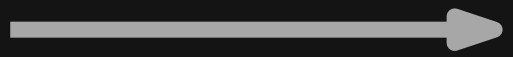


Where to fix the bug?

Multi-million file **code base**

Motivation

Thousands of field crashes every day



Where to fix the bug?

Multi-million file code base

Relevant for

- Finding the right team or developer
- Giving the developer a starting point
- Enabling automated program repair

Goal

Crash-based, file-level bug localization

Goal

Crash-based, file-level bug localization

Only input:
Raw crash trace

Goal

Crash-based, file-level bug localization

Only input: Raw crash trace

Desired output: File where to fix the bug

Example of Crash Trace

Nb.	Lines of raw crash trace
1	`android_crash:java.lang.IllegalStateException: ...`
2	`stack_trace:  java.lang.RuntimeException: Unable to ...`
3	`at android.app...(ActivityThread.java:3975)`
...	**(dozens of more lines)**
23	`Caused by: java.lang.IllegalStateException: Activity has ...`
24	`at android.app.FragmentManagerImpl.enqueueAction...`
25	`at android.app...commitInternal(BackStackRecord.java:745)`
26	`at android.app...commitAllowingStateLoss(...)`
27	`at ...clearBrowserFragment(DoStuffBrowserController.java:775)`
28	`at ...exitDoMoreStuff(DoStuffBrowserController.java:196)`
29	`at ...DoMoreStuffRootView.exitDoMoreStuff(...)`
...	**(dozens of more lines)**
76	`app_upgrade_time: 2018-08-19T17:02:12.000+08:00`
77	`package_name: lala`
78	`peak_memory_heap_allocation:  92094532`
79	`app_backgrounded: false`
...	**(dozens of more lines)**

Example of Crash Trace (2)

Nb.	Lines of raw crash trace
1	`[twi01447.08.ftw1.example.com] [Sat Sep 29 13:20:40 2018] ...`
2	`(Events: <null_response_query_id>)`
3	`(App Version: 189.0.0.44.93)`
4	`(NNTraceID: FAiHFWy4Isj)`
5	`(Sampling ID: A_oSj-oZbo1GJnM8JHKL3o_)`
...	**(dozens of other lines)**
31	`trace starts at [/var/abc/def/core/runtime/error.php:1021]`
32	`#0 __log_helper(...) called at [/var/abc/.../error.php:1021]`
33	`#1 log() called at [/var/abc/def/core/logger/logger.php:1162]`
34	`#2 NNDefaultLogMessage->log() called at [/var/abc/...:938]`
35	`#3 NNLogMessage->process() called at [/var/abc/def/...]`
36	`#4 NNLogMessage->warn() called at [/var/abc/def/...]`
37	`#5 NNLogMessage->warn_High() called at [/var/abc/...]`
38	`#6 MultifooClient->genPopulateResults$memoize_impl() ...]`
39	`#7 ... called at [/var/abc/.../MultifooQuery.php:5782]`
40	`#8 Closure$MultifooQuery::genStuff#9() called at...]`
...	**(dozens of other lines)**

Challenges

Scalability

Heterogeneous code base

Fuzzy information

Challenges

Scalability

Code base with multiple millions of files

Heterogeneous code base

Fuzzy information

Challenges

Scalability

Code base with multiple millions of files

Heterogeneous code base

Different languages, platforms, and domains

Fuzzy information

Challenges

Scalability

Code base with multiple millions of files

Heterogeneous code base

Different languages, platforms, and domains

Fuzzy information

Information in crash traces may not exactly match code base

Prior Work on Bug Localization

Based on traces of correct and buggy executions

[Abreu2007, Jones2002, Li2018]

Based on evidence of a bug, e.g., stack trace or bug report

[Nguyen2011, Zhou2012, Saha2013, Kim2013, Rongxin2014, Ye2014, Lam2015, Koyuncu2019]

Prior Work on Bug Localization

Based on traces of correct and buggy executions

[Abreu2007, Jones2002, Li2018]

Needs tests that reproduce field crash

Based on evidence of a bug, e.g., stack trace or bug report

[Nguyen2011, Zhou2012, Saha2013, Kim2013, Rongxin2014, Ye2014, Lam2015, Koyuncu2019]

Prior Work on Bug Localization

Based on traces of correct and buggy executions

[Abreu2007, Jones2002, Li2018]

Needs tests that reproduce field crash

Based on evidence of a bug, e.g., stack trace or bug report

[Nguyen2011, Zhou2012, Saha2013, Kim2013, Rongxin2014, Ye2014, Lam2015, Koyuncu2019]

Scalability problems

Prior Work on Bug Localization

Based on traces of correct and buggy executions

[Abreu2007, Jones2002, Li2018]

Needs tests that reproduce field crash

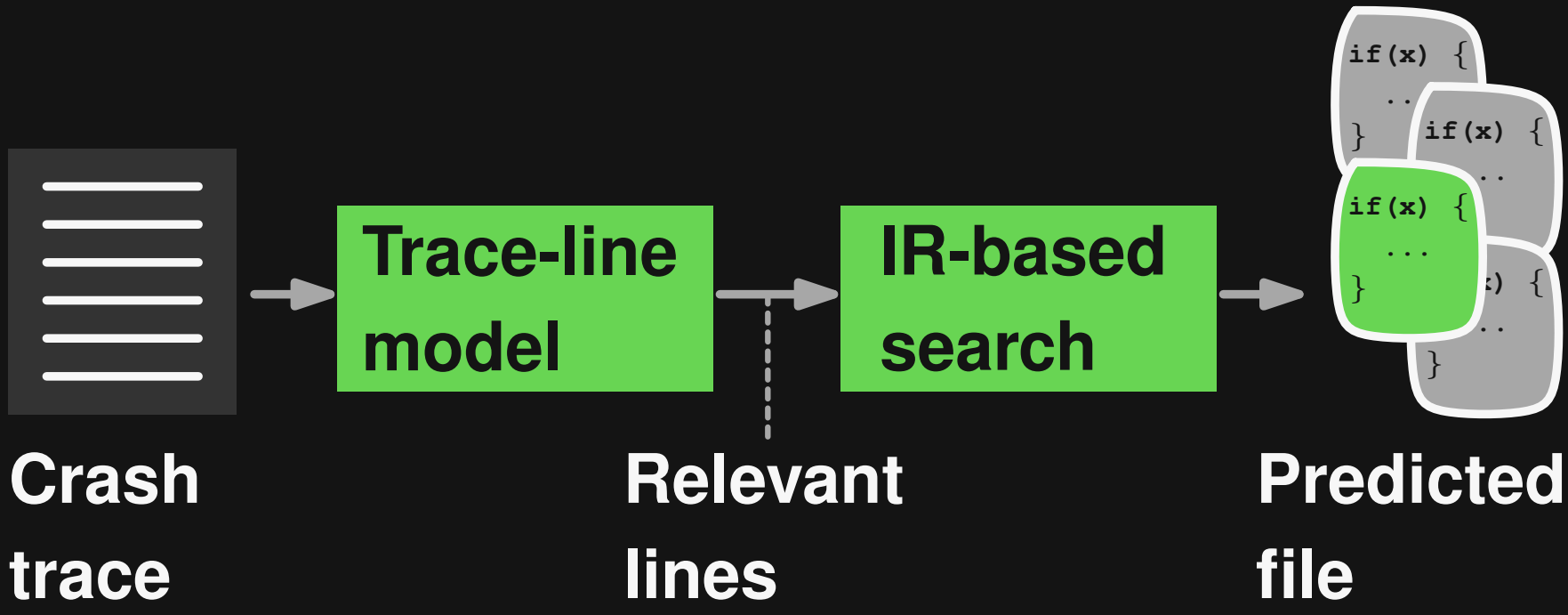
Based on evidence of a bug, e.g., stack trace or bug report

[Nguyen2011, Zhou2012, Saha2013, Kim2013, Rongxin2014, Ye2014, Lam2015, Koyuncu2019]

Scalability problems

Focus on a single programming language

Scaffle: Overview

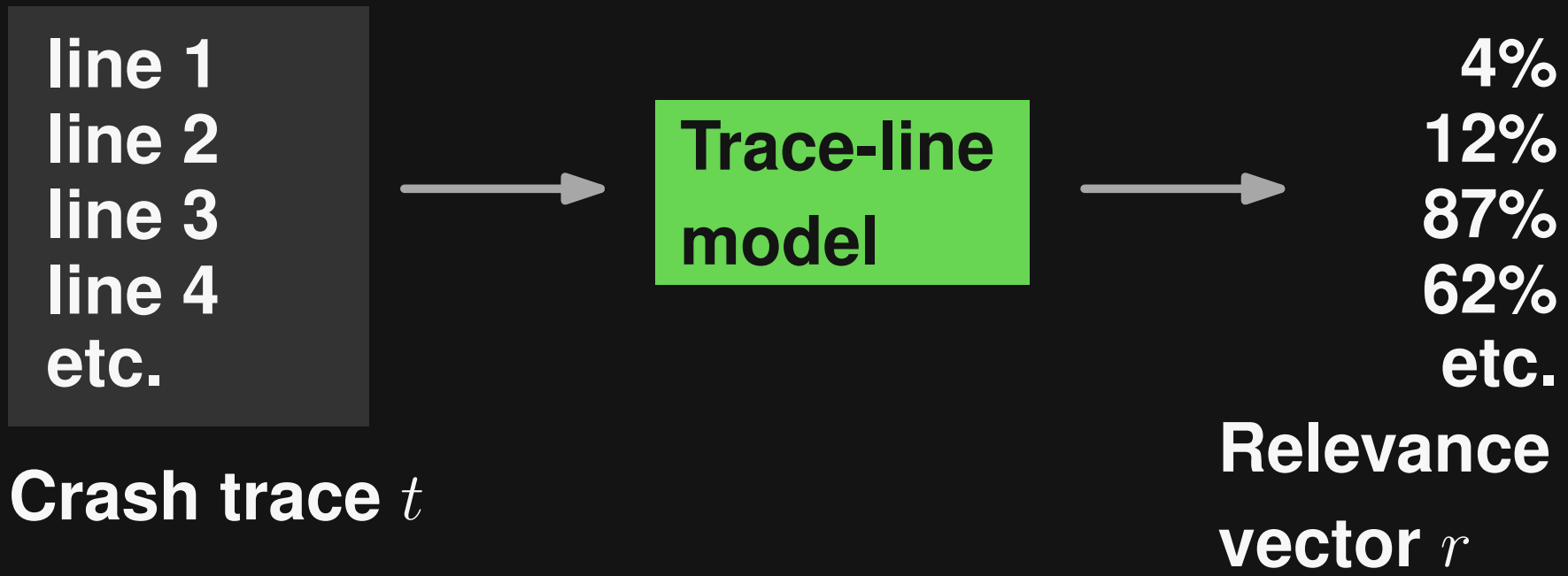


Key insight:

Decomposition of the problem

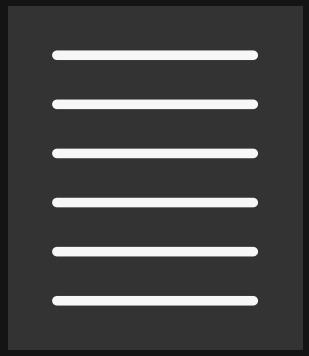
Trace-line Model

Machine learning model:
Predict relevance of lines in crash trace

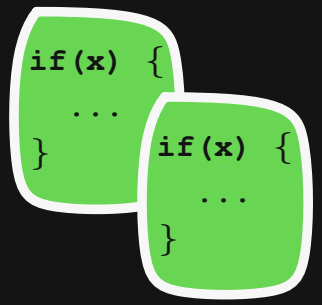


Training Data from Past Crashes

Compute pairs (t, r) of traces and relevance vectors from fixed crashes



Crash trace



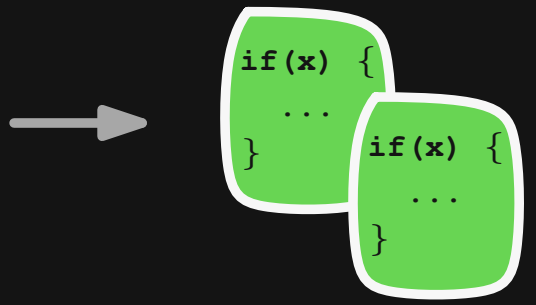
Files changed in crash fix

Training Data from Past Crashes

Compute pairs (t, r) of traces and relevance vectors from fixed crashes

Crash trace

Which lines are most relevant to locate the fixed files?



Files changed in crash fix

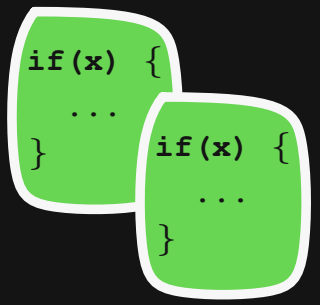
Training Data from Past Crashes

Compute **pairs** (t, r) **of traces and relevance vectors from fixed crashes**

Crash trace

Tokens in lines

Segments in file paths



Files changed in crash fix

More **overlap**

$\Rightarrow$ More **relevant**

Example of Crash Trace

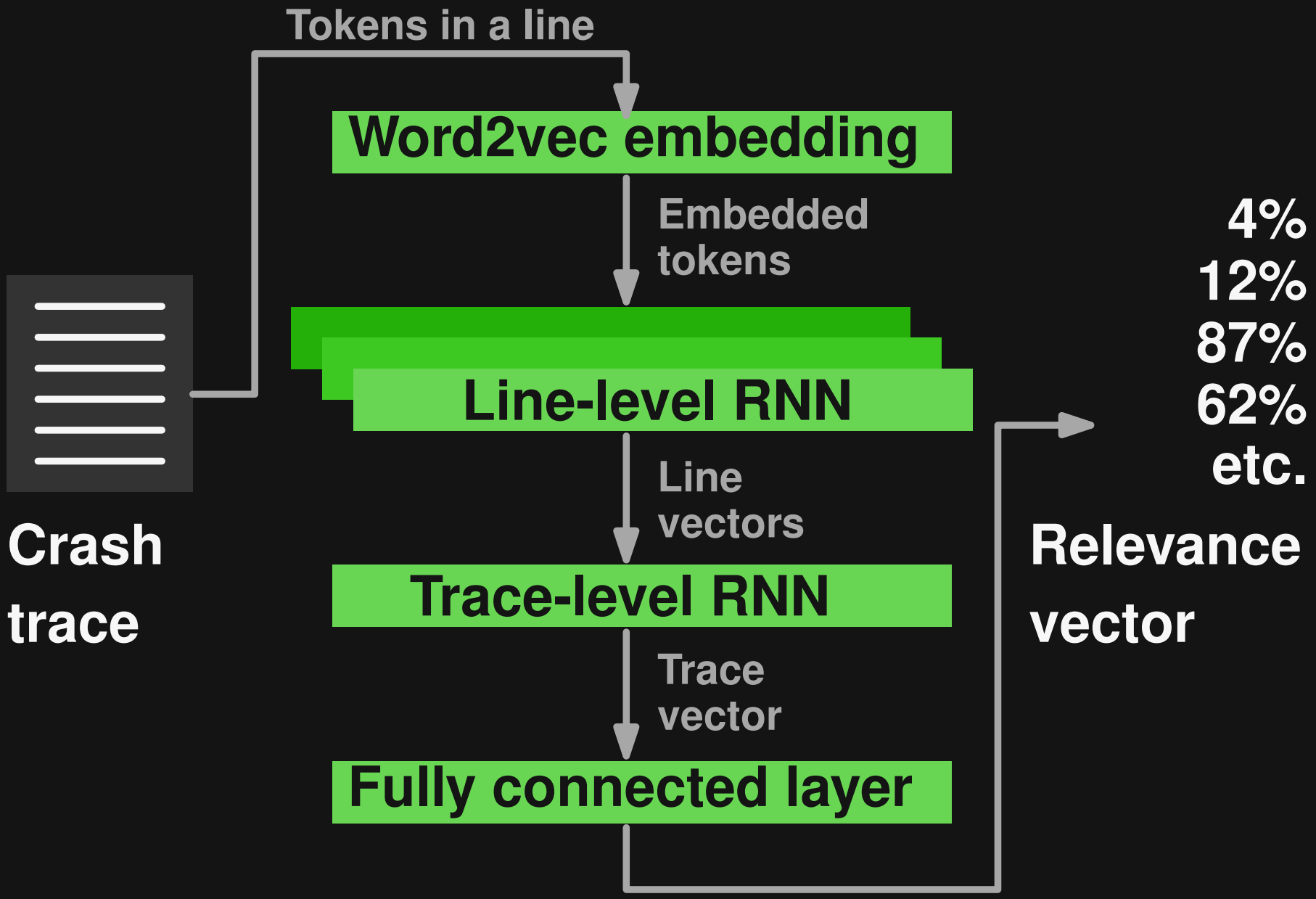
Nb.	Lines of raw crash trace
1	`android_crash:java.lang.IllegalStateException: ...`
2	`stack_trace:  java.lang.RuntimeException: Unable to ...`
3	`at android.app...(ActivityThread.java:3975)`
...	**(dozens of more lines)**
23	`Caused by: java.lang.IllegalStateException: Activity has ...`
24	`at android.app.FragmentManagerImpl.enqueueAction...`
25	`at android.app...commitInternal(BackStackRecord.java:745)`
26	`at android.app...commitAllowingStateLoss(...)`
27	`at ...clearBrowserFragment(DoStuffBrowserController.java:775)`
28	`at ...exitDoMoreStuff(DoStuffBrowserController.java:196)`
29	`at ...DoMoreStuffRootView.exitDoMoreStuff(...)`
...	**(dozens of more lines)**
76	`app_upgrade_time: 2018-08-19T17:02:12.000+08:00`
77	`package_name: lala`
78	`peak_memory_heap_allocation:  92094532`
79	`app_backgrounded: false`
...	**(dozens of more lines)**

Example of Crash Trace

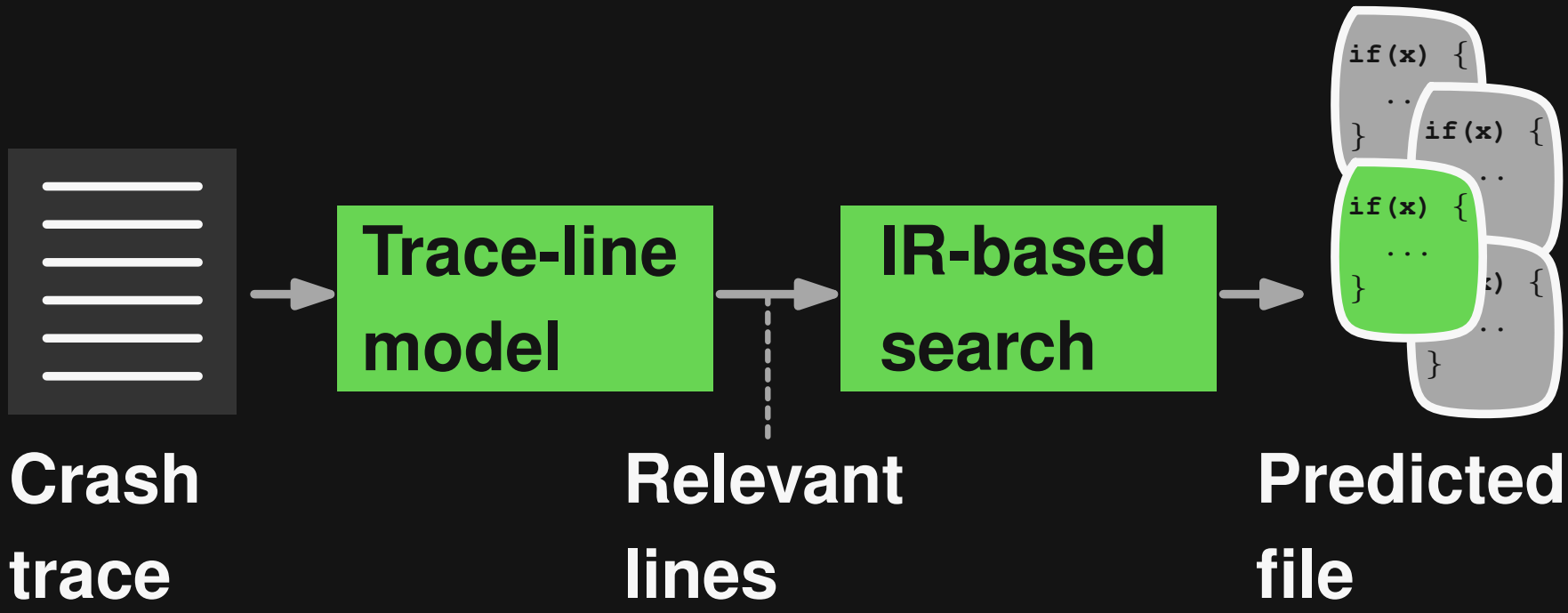
Nb.	Lines of raw crash trace
1	`android_crash:java.lang.IllegalStateException: ...`
2	`stack_trace:  java.lang.RuntimeException: Unable to ...`
3	`at android.app...(ActivityThread.java:3975)`
...	**(dozens of more lines)**
23	`Caused by: java.lang.IllegalStateException: Activity has ...`
24	`at android.app.FragmentManagerImpl.enqueueAction...`
25	`at android.app...commitInternal(BackStackRecord.java:745)`
26	`at android.app...commitAllowingStateLoss(...)`
27	`at ...clearBrowserFragment(DoStuffBrowserController.java:775)`
28	`at ...exitDoMoreStuff(DoStuffBrowserController.java:196)`
29	`at ...DoMoreStuffRootView.exitDoMoreStuff(...)`
...	**(dozens of more lines)**
76	`app_upgrade_time: 2018-08-19T17:02:12.000+0`
77	`package_name: lala`
78	`peak_memory_heap_allocation:  92094532`
79	`app_backgrounded: false`
...	**(dozens of more lines)**

Most relevant lines if bug fixed in DoStuffBrowser-Controller.java

Neural Trace Line Model



Scaffle: Overview



Key insight:

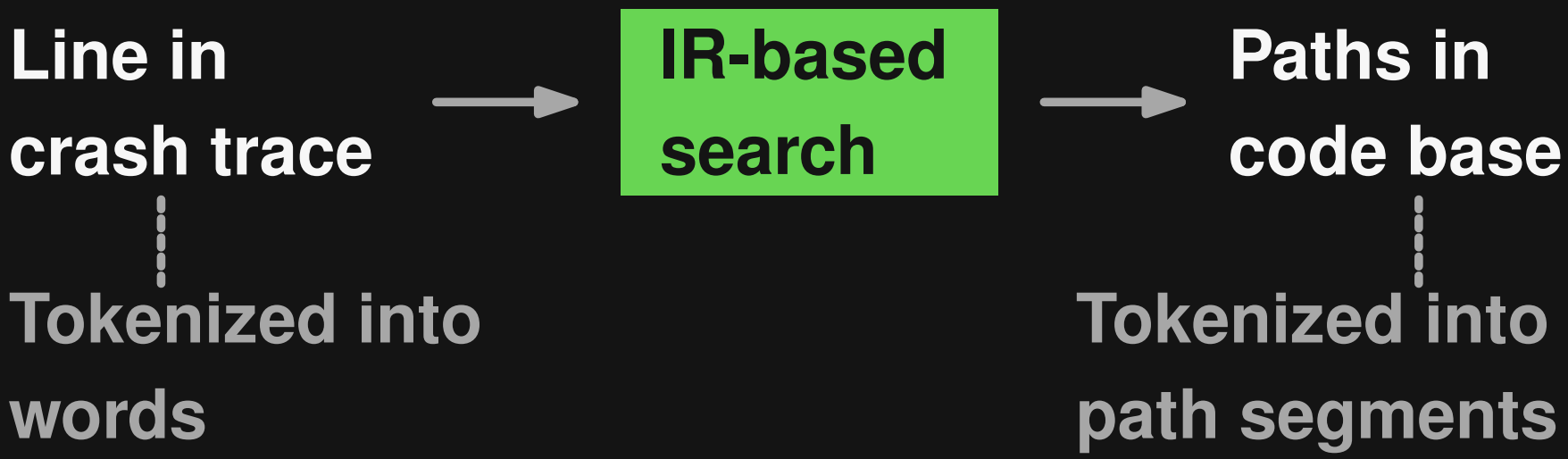
Decomposition of the problem

IR-based Search for Bug Location

IR-based Search for Bug Location

IR-based Search for Bug Location

IR-based Search for Bug Location



Example:
“at exitDoMoreStuff DoStuffBrowserController java”

Example:
“projectX/packageA/-dostuff/DoStuffBrowser-Controller.java”

Evaluation

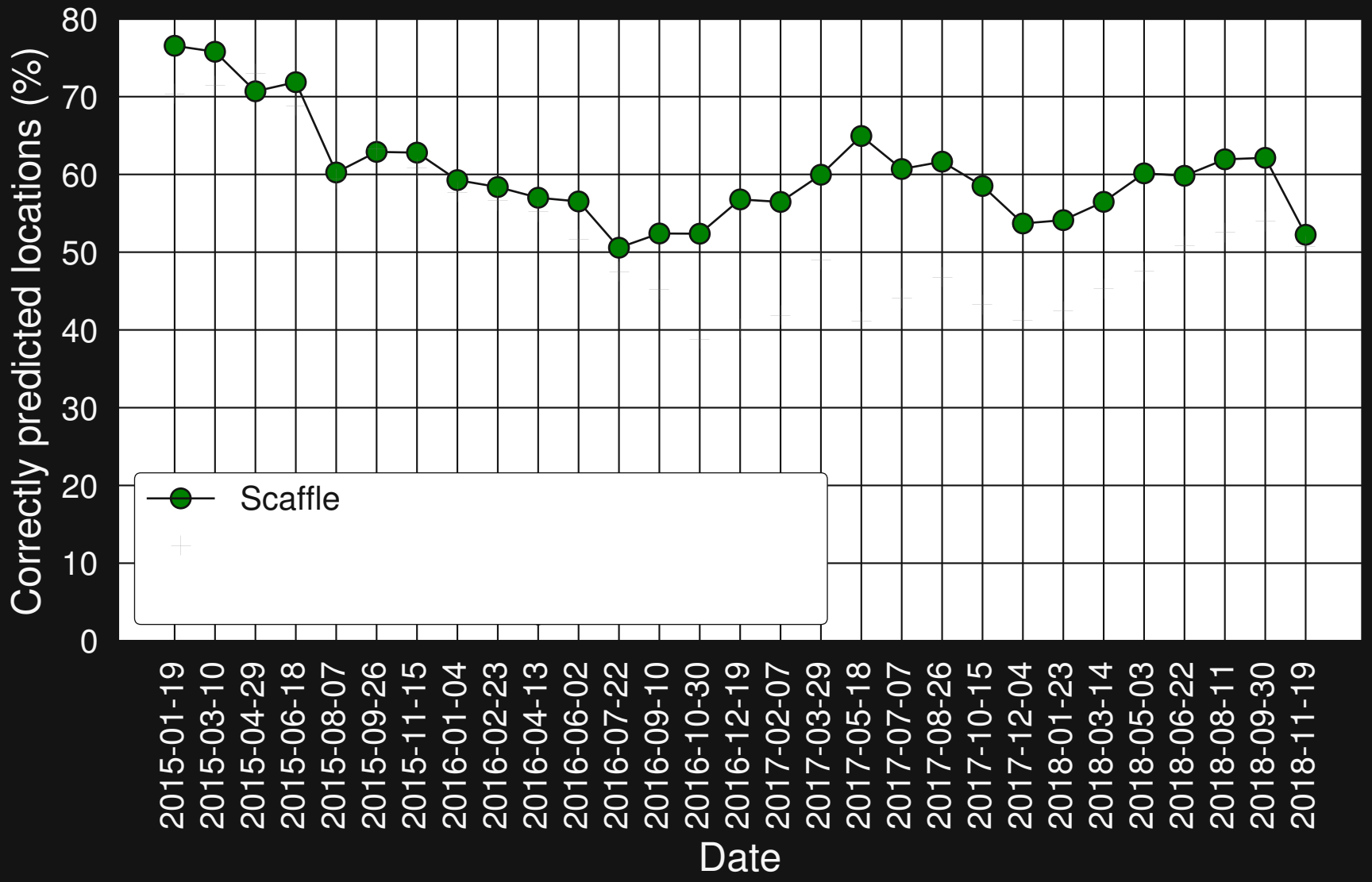
≈ **20k crashes** of various Facebook products (Android, iOS, PHP)

Code base: **Monorepo of Facebook**

4-year period, 50-day steps

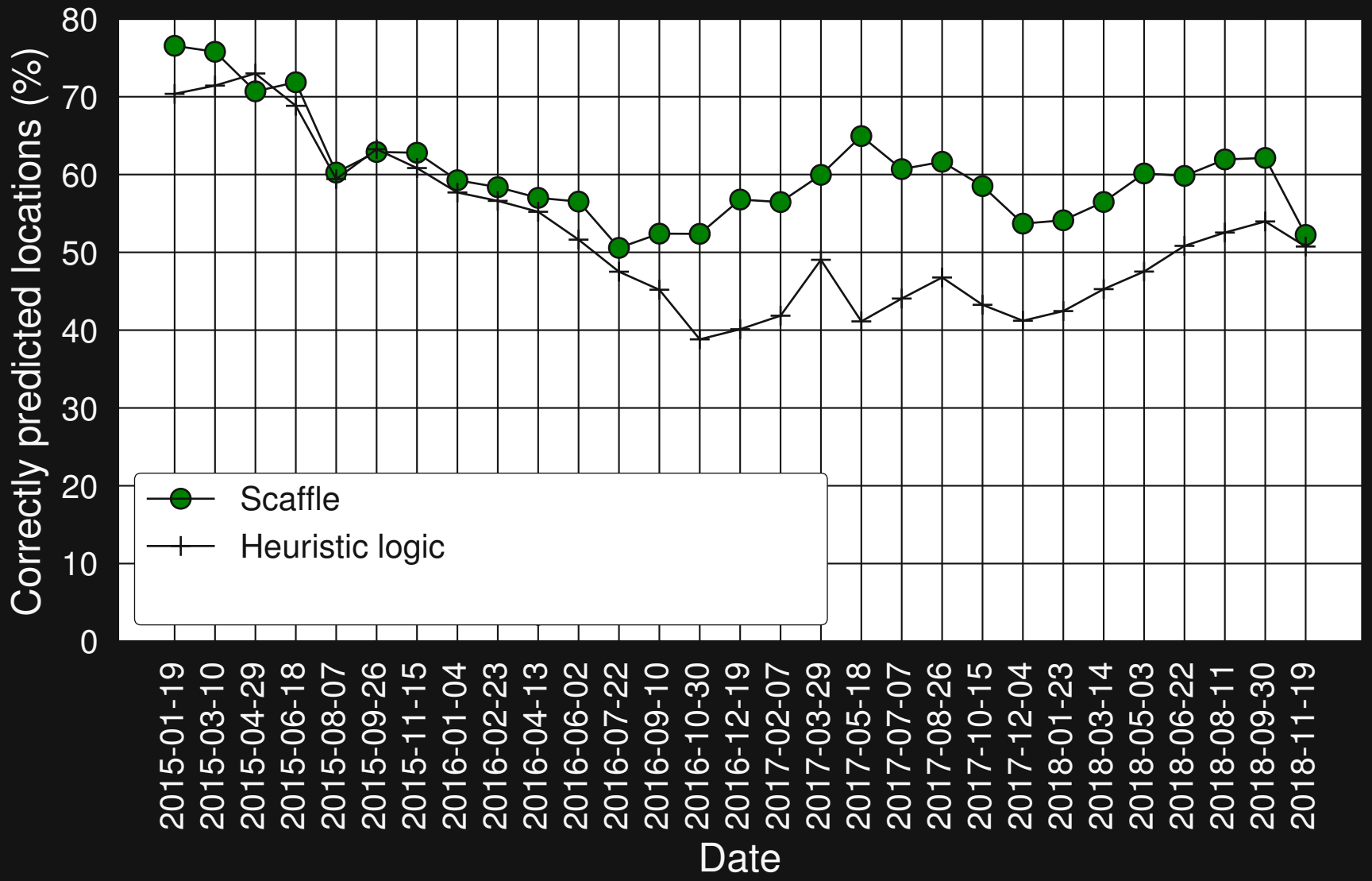
End-to-End Effectiveness

Top-5 predictions on raw crash traces



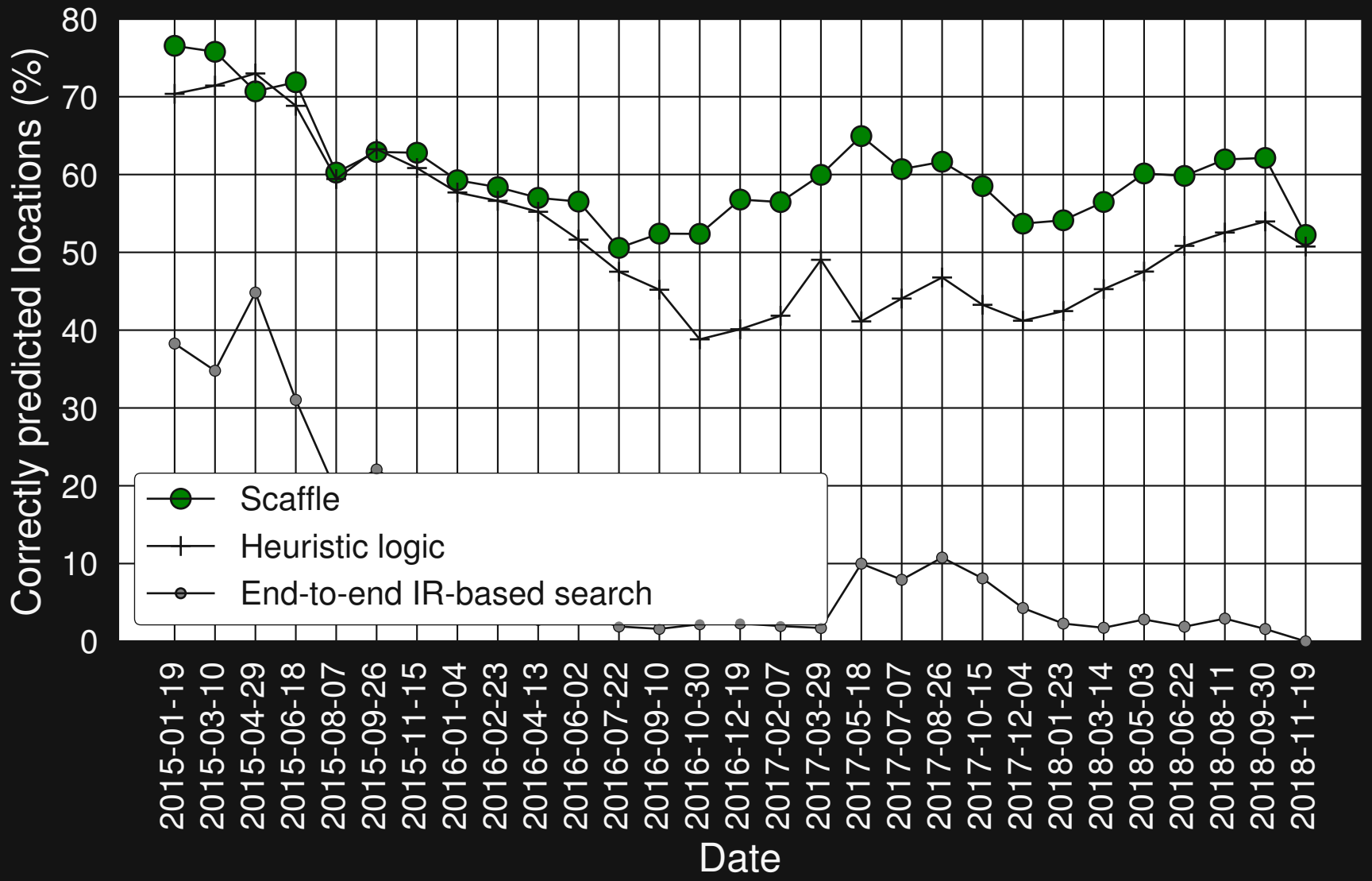
End-to-End Effectiveness

Top-5 predictions on raw crash traces



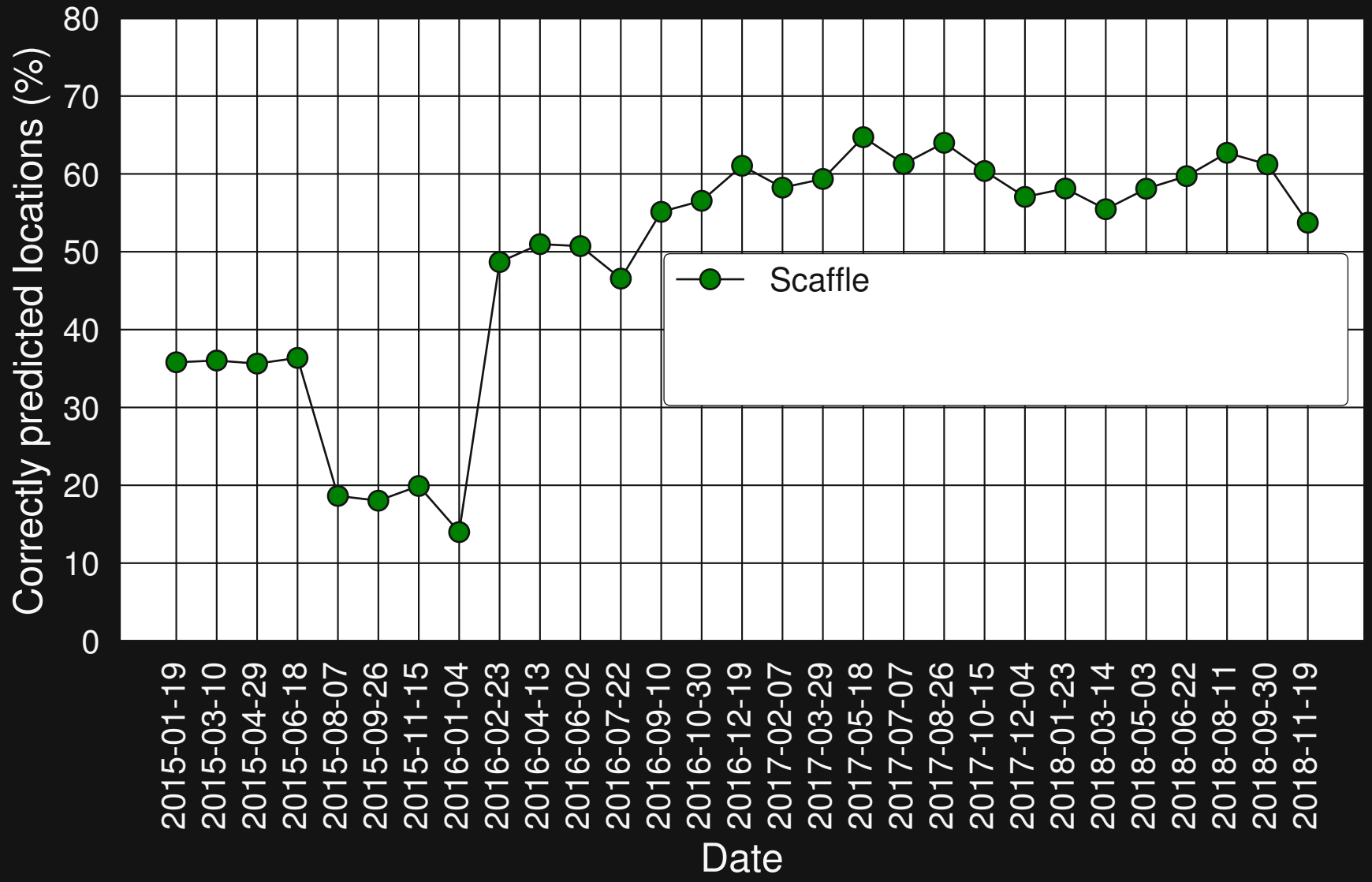
End-to-End Effectiveness

Top-5 predictions on raw crash traces



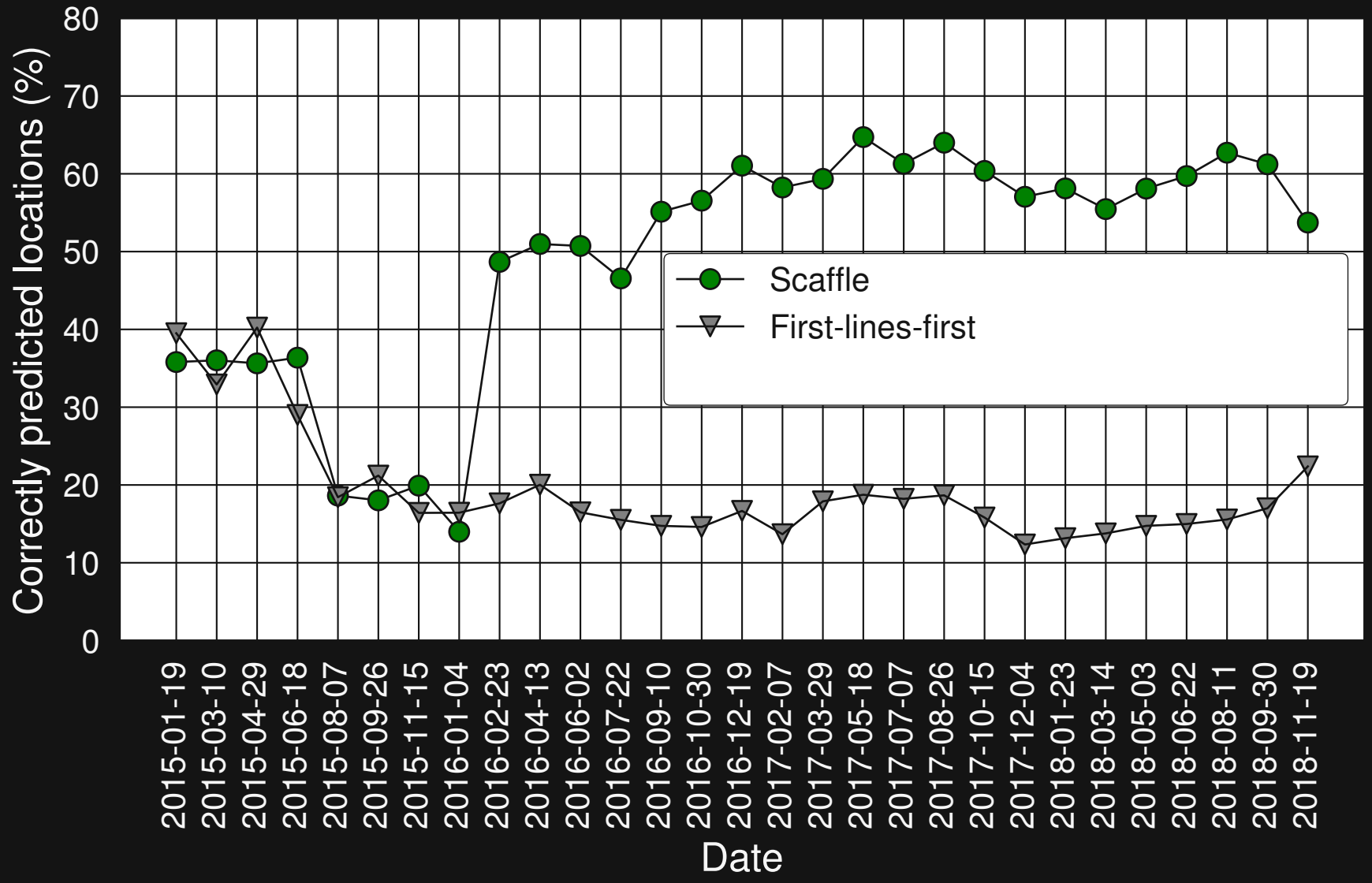
End-to-End Effectiveness (2)

Top-5 predictions on stack traces only



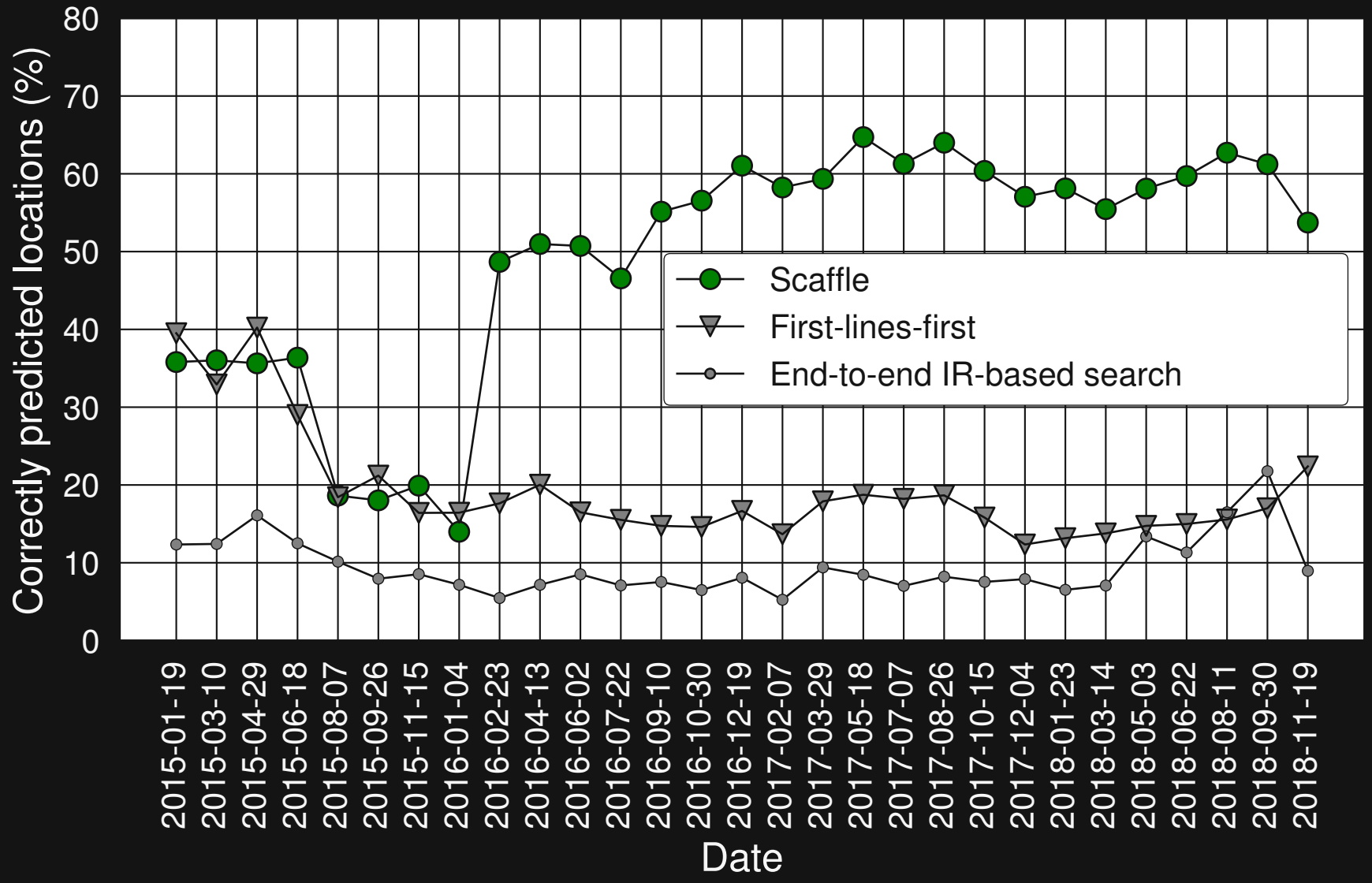
End-to-End Effectiveness (2)

Top-5 predictions on stack traces only



End-to-End Effectiveness (2)

Top-5 predictions on stack traces only

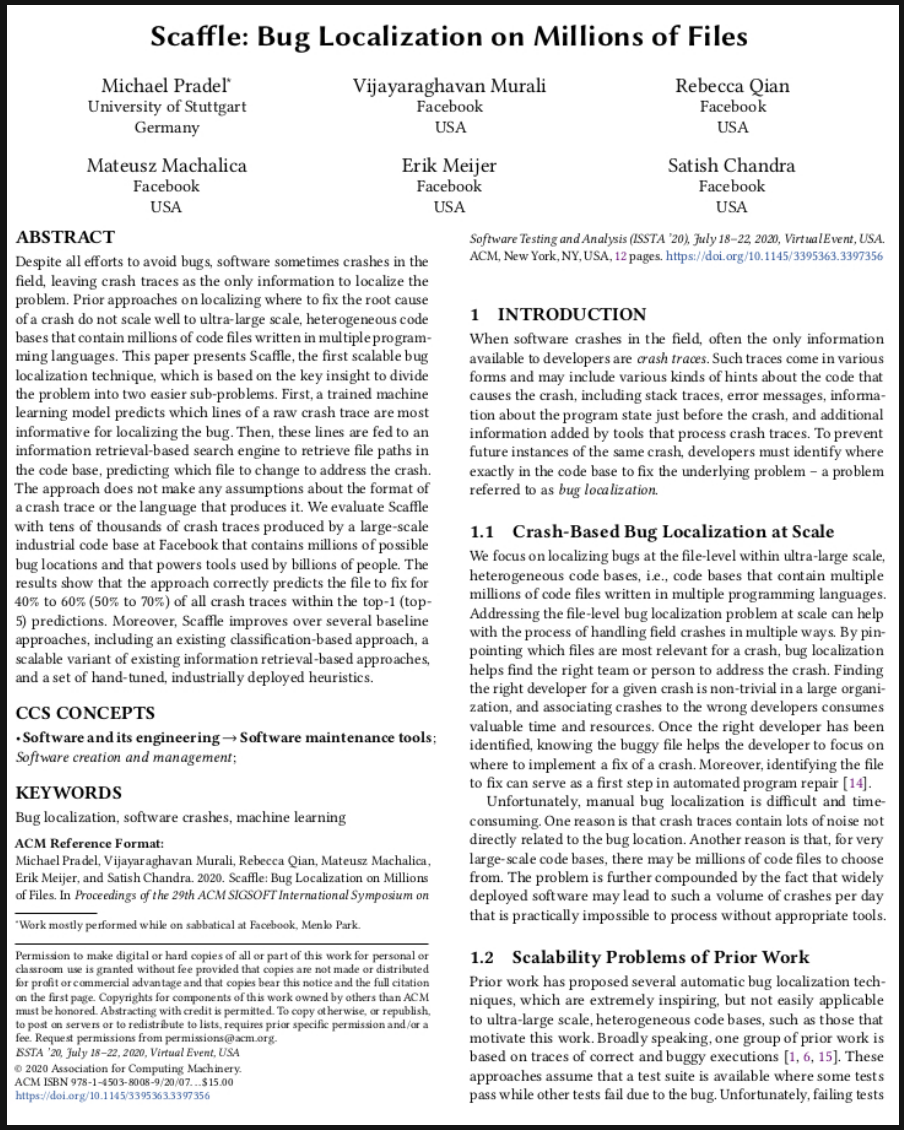


Why Does It (Sometimes Not) Work?

- Buggy files (not) mentioned in traces
- Can(not) handle partial information about files
- (No) understanding of the structure of crash traces

More Results: Paper

- Details on dataset
- Effectiveness of trace-line model
- Efficiency
- Detailed comparison with baselines



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

Conclusions

Scaffle:

Bug localization on millions of files

- New decomposition of the problem
  - Identify most relevant lines in a trace
  - Match relevant lines against paths in code base
- Learning-based, language-independent:

  Easy to adapt to evolving code bases