DEFAULT: Mutual Information-based Crash Triage for Massive Crashes

Xing Zhang[1], Jiongyi Chen[1(✉)], Chao Feng[1], Ruilin Li[1],
Wenrui Diao[2,3], Kehuan Zhang[4], Jing Lei[1], Chaojing Tang[1]

[1]National University of Defense Technology
[2]School of Cyber Science and Technology, Shandong University
[3]Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University
[4]Chinese University of Hong Kong

ABSTRACT

With the considerable success achieved by modern fuzzing infrastructures, more crashes are produced than ever before. To dig out the root cause, rapid and faithful crash triage for large numbers of crashes has always been attractive. However, hindered by the practical difficulty of reducing analysis imprecision without compromising efficiency, this goal has not been accomplished.

In this paper, we present an end-to-end crash triage solution DEFAULT, for accurately and quickly pinpointing unique root cause from large numbers of crashes. In particular, we quantify the "crash relevance" of program entities based on mutual information, which serves as the criterion of unique crash bucketing and allows us to bucket massive crashes without pre-analyzing their root cause. The quantification of "crash relevance" is also used in the shortening of long crashing traces. On this basis, we use the interpretability of neural networks to precisely pinpoint the root cause in the shortened traces by evaluating each basic block's impact on the crash label. Evaluated with 20 programs with 22216 crashes in total, DEFAULT demonstrates remarkable accuracy and performance, which is way beyond what the state-of-the-art techniques can achieve: crash de-duplication was achieved at a super-fast processing speed – 0.017 seconds per crashing trace, without missing any unique bugs. After that, it identifies the root cause of 43 unique crashes with no false negatives and an average false positive rate of 9.2%.

CCS CONCEPTS

• **Security and privacy** → *Software security engineering*.

KEYWORDS

Crash Triage; Software Security

✉ Corresponding author, chenjiongyi@nudt.edu.cn.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3512760

ACM Reference Format:
Xing Zhang, Jiongyi Chen, Chao Feng, Ruilin Li, Wenrui Diao, Kehuan Zhang, Jing Lei, and Chaojing Tang. 2022. DEFAULT: Mutual Information-based Crash Triage for Massive Crashes. 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/10.1145/3510003.3512760

1 INTRODUCTION

Software vulnerability is a prevailing threat in cyberspace. To discover and eliminate software vulnerabilities, fuzzing has been recognized as one of the most effective approaches by randomly or strategically generating a large number of inputs to feed a program and trigger program exceptions. For example, the fuzzing infrastructure ClusterFuzz has found more than 25,000 bugs in Google products (e.g., Chrome) and around 22,500 bugs in over 340 open source projects in September 2020 [3]. Even though considerable progress has made in triggering crashes, the subsequent procedure—crash triage—remains imprecise, time-consuming, and labor-intensive.

Accuracy, efficiency, and generality are the major concerns of current crash triage techniques. However, there lacks a systematic solution that can balance the trade-offs and achieve accurate, fast, and fully-automated crash triage. For the past decade, although there has been a wealth of research into crash triage, including crash deduplication [16, 18, 27, 29, 33, 34] and fault localization [5, 10, 20, 25, 26, 31, 42, 44, 47, 48], the efficacy of those approaches significantly varies based on different vulnerability types, crash reports, and running environment, making them less applicable to general programs. In particular, a line of prior crash deduplication approaches work at the granularity of function call level and fail to bucket crashes by inspecting the crashes' actual root cause, which could cause critical vulnerabilities triggered but missed. The other research aims to deduplicate crashes by examining the root cause of crashes or representing root cause with the constraints on crashing paths, which takes significant time when the number of processed crashes increases. Regarding fault localization, given that prior statistical approaches fail to capture sequence information of root-cause basic blocks, the accuracy of identification is seriously affected. More importantly, those approaches only output suspicious scores for a set of basic blocks, for example, top 20 basic blocks. In practice, this does not give sufficient guidance for analysts, as they still have a set of basic candidate blocks to examine.

In this paper, we present an end-to-end crash triage solution called DeFault, for rapidly de-duplicating massive traces of crashes and accurately pinpointing the root cause for unique crashes. We borrow the concept of mutual information in information theory and treat crash triage as an information mining process. The key insight is that mutual information of program entities (e.g., functions and basic blocks) is a measurement of their relevance to the crash. We leverage such "crash relevance": (1) as the criteria to bucket unique crashes in crash deduplication; (2) to identify and filter out irrelevant program entities for shortening traces in our neural network-based fault localization. Without inspecting each crash's root cause, our approach allows crash deduplication accomplished within a short time without missing bugs. In the subsequent fault localization procedure, the mutual information about "crash relevance" is used again to filter program entities that are irrelevant to the crash, which shortens the long execution traces and thus facilitates the feeding of inputs to the neural network. On such a basis, in the last step, we utilize the interpretability of neural networks to extract the actual root cause from the shortened traces. With the ability to capture sequence information, the neural network significantly improves root cause identification accuracy compared with the traditional approaches.

We implemented a full-featured prototype of DeFault and evaluated it with 20 programs, including 8 CGC programs and 12 real-world programs. On the one hand, for crash de-duplication, DeFault processed 22216 crashing traces at a speed of 0.017 seconds per trace. It identified 42 unique crashing traces without missing any bugs. Regarding fault localization, it reports no false negatives and low false positive rate of 9.2%, which is way beyond what the state-of-the-art tools can achieve.

Contributions. The contributions of this paper are summarized as follows.

- **New techniques.** We propose a new approach to measure the "crash relevance" of program entities based on mutual information, which is critical for crash deduplication and fault localization. With such an approach, we design and present a novel end-to-end analysis system, DeFault, that directly takes crashing execution traces from fuzzers as input and automatically pinpoints the root cause of program faults.
- **Evaluation.** We evaluated DeFault on 20 programs, including 8 CGC programs and 12 real-world programs. The results demonstrate that DeFault is both efficient and accurate. The tool publicly available for continuous research[1].

Roadmap. The rest of this paper is organized as follows: Section 2 surveys the related research. Section 3 provides the necessary background, covering mutual information and the attention mechanism of interpretability of neural networks. Section 4 describes the detailed design of DeFault. Section 5 presents the evaluation results and the comparison with existing techniques. Section 7 concludes this paper.

[1]DeFault is available at https://github.com/zxhree/default

2 RELATED WORK

In this section, we review the related work on crash deduplication and fault localization for software testing. The limitations of the existing approaches are also summarized.

2.1 Crash Deduplication

Crash deduplication aims to cluster crashes produced by fuzzers and select a unique crash (or a representative crash) from each clustered group of crashes that share the same root cause. The subsequent fault localization is performed on such unique crashes. With more crashes produced by fuzzing infrastructures, crash deduplication has become an urgent demand, easing the burden of subsequent fault localization. However, existing approaches are far from accurate and practical, as they are either coarse-grained or make certain assumptions. Below we describe the related work.

Call stack-based deduplication. The widely used call stack-based approaches measure the similarity among function call sequences [16, 18, 27, 29], function arguments [9], or call graphs [23] of the functions on call stacks. In general, such approaches are coarse-grained. If a program with different vulnerabilities crashed in the same function, call stack-based deduplication would miss unique crashes.

Constraint-based deduplication. As the typical representative, Pham et al. [33] and Podelski et al. [34] collect constraints on the failing paths and passing paths, and deduce their longest common prefix, which is further used to characterize the semantics of the failure. deduplication is conducted based on the unique symbolic semantics. However, hindered by the drawbacks of symbolic execution techniques [11, 14] (e.g., control flow dependence and unsolvable constraints), such approaches are less scalable for massive crashes of real-world programs.

Patching-based deduplication. Patching-based deduplication first automatically fixes specific vulnerabilities [12, 28, 37] (e.g., buffer overflow and null pointer dereference) at crashing point and then observes whether unfixed crashes can be reproduced. The crashes that cannot be reproduced are clustered into the same group. This approach relies on source code analysis and specific vulnerability types, not to mention the side effect of program transformation. Therefore, it is less applicable to binary programs with unknown vulnerability types.

Report-based deduplication. Report-based deduplication leverages information retrieval techniques to analyze text information of crash reports. For examples, Wang et al. [38] leverage topic models, Scaffle et al. [35] use neural networks, and Ye et al. [46] utilize sort algorithms, to extract crash-related information like call sequences and the history of vulnerability patching. Kim et al. [22] use machine learning techniques to predict the root cause, for the purpose of differentiating various crashes. However, report-based approaches are too coarse, which are typically performed upon the function level. Apart from that, the effectiveness depends on how the OS or the debugger writes the record, which may not provide sufficient crash-related information.

2.2 Fault Localization

Automatic fault localization techniques leverage the statistics of target programs at runtime to locate program faults. The related work can be categorized as follows:

Program spectrum-based approaches. Program spectrum, such as the statistics of program paths executions, is a measurement of program running status. Collefello et al. [15] for the first time leveraged program spectrum for fault localization. By comparing the statistics of program entities about normal exit and crash, a suspicious score is given to each program entity. Generally, the more execution time a program entity has in crash samples than normal exit samples, the higher score the program entity gets. The scores of program entities can be calculated and ranked according to various approaches [5, 10, 19, 20, 25, 26, 31, 42, 44, 47, 48]. However, all existing approaches only consider whether a program entity exists in samples but neglect the execution times of the entities in a certain sample and the sequence of their executions. As demonstrated in Section 5, without such sequence information, program spectrum-based fault localization would inevitably introduce imprecision.

Machine learning-based approaches. Machine learning-based approaches [30, 32, 36, 50] treat program entities as input and leverage the classification systems to output the probability of root cause for each program entity. For instance, Liu et al. [30] represent the execution flow of a program as a graph. Also, they use graph mining approaches to extract sub-graphs about program faults and use SVMs to identify each sub-graph's contributions to the crash. Similarly, Nessa et al. [32] use N-Gram to calculate the conditional probability of program entities in execution traces to the crash. However, similar to program spectrum-based approaches, those studies do not recover sequence information for root-cause basic blocks.

Program slicing-based approaches. Program slicing techniques [39], including static slicing and dynamic slicing, are to compute a set of program statements that may affect the values at some point of interest. In particular, dynamic slicing [6] is often used in fault localization by analyzing program execution traces. Agrawal et al. [7] leverage dynamic slicing to locate program faults by calculating the slices' intersection. Wang et al. [40, 41] use dynamic slicing to pinpoint program faults by analyzing data dependence among program entities. Zhang et al. [51, 53] use dynamic slicing to extract the change of variables related to the crash to locate root causes. Moreover, Xu et al. [43] use backward program slicing to locate program faults with the support of Intel PT. However, due to the control flow dependence problem – a fundamental drawback of dataflow analysis, the effectiveness of those approaches is not satisfactory.

2.3 Limitations of Prior Research

Though much effort has been put into the research of crash triage, the existing approaches still have multiple critical limitations, as summarized below.

- **Generality.** The effectiveness of crash triage techniques significantly varies with different vulnerability types, crash reports, and running environment, making them less applicable to general programs.
- **Accuracy.** Prior crash deduplication approaches work at the granularity of function call level and fail to bucket crashes by inspecting the actual root cause of the crashes, which may cause imprecision and miss bugs. On the other hand, existing fault localization approaches do not consider sequence information, which introduces imprecision again.
- **Time consumption.** To bucket the crashes, existing approaches examine each crash regardless of the granularity. After that, a pair-wise match among all crashes is unavoidable, which takes significant time with the number of crashes increasing.

3 PRELIMINARIES

This section provides some necessary backgrounds of mutual information and introduces the attention mechanism of neural networks.

3.1 Mutual Information

In information theory, mutual information is a measure of the mutual dependence between two random variables. It can be regarded as the amount of information that one random variable contains about the other random variable. Mutual information is an important criterion for feature selection in machine learning. The more information a feature brings to the classification system, the larger the value of its mutual information is. Namely, such a feature is more relevant to the classification. Inspired by the fact, we leverage the mutual information to measure the contribution of basic blocks to crashes. We use the mutual information not only in crash de-duplication, but also in the initial step of fault localization by filtering out the vast majority of program entities (e.g., functions and basic blocks) that are irrelevant to the crash.

Mutual information. We represent a crashing trace at basic block level as a list of tuples: $\mathbf{B} = \{b_1 : n(b_1); b_2 : n(b_2)...; b_n : n(b_n)\}$, where b is (the starting address of) a basic block, $n(b)$ is its occurrence in trace $\mathbf{B}$. The output of a fuzzer forms a dataset $\mathbb{D} =< \mathbb{B}, \mathbb{Y} >$, where $\mathbb{B}$ is a set of crashing traces $\{\mathbf{B}_1, \mathbf{B}_2, ..., \mathbf{B}_n\}$ and $\mathbb{Y} = \{y_1, y_2, ..., y_n\}$ is a set of labels denoting whether a trace corresponds a crash or a normal exit.

For a given dataset D, the mutual information of a basic block b to label y is given by:

$$I(y|b) = H(b) - H(b|y) = H(y) - H(y|b) \tag{1}$$

where $H(y)$ is the entropy of label y and $H(y|b)$ is the conditional entropy of basic block b given label y. More specifically, we denote N_p as the amount of non-crashing traces in dataset D and N_f as the amount of crashing traces in dataset D (where $n = N_f + N_p$). Then the entropy of label y is:

$$H(y) = -\frac{N_p}{N} log(\frac{N_p}{N}) - \frac{N_f}{N} log(\frac{N_f}{N}) \tag{2}$$

Using $max(n(b))$ to denote the max of $n(b)$ for basic block b in all $\mathbf{B}$ that belong to D, we have the conditional entropy $H(y|b)$:

$$H(y|b) = -\sum_{i=0}^{max(n(b))} p(b_{|_{n(b)=i}}) H(y|b_{|_{n(b)=i}}) \tag{3}$$

where $p(b_{|_{n(b)=i}}) = \frac{c_i(b)}{N}$.

Additionally, the conditional entropy of y to $b|_{n(b)=i}$ is:

$$H(y|b|_{n(b)=i}) = -\frac{c_{pi}(b)}{c_i(b)}log(\frac{c_{pi}(b)}{c_i(b)}) - \frac{c_{fi}(b)}{c_i(b)}log(\frac{c_{fi}(b)}{c_i(b)}) \quad (4)$$

In the above formula, $c_i(b)$ is the amount of **B** in D when $n(b) = i$, $c_{pi}(b)$ is the amount of **B** in D when $y = 0$ and $n(b) = i$, and $c_{fi}(b)$ is the amount of **B** in D when $y = 1$ and $n(b) = i$. Combining Equation(3) and Equation (4), we can obtain the conditional entropy $H(y|b)$ and the mutual information $I(y|b)$:

$$H(y|b) = -\sum_{i=0}^{max(n(b))} \frac{c_i(b)}{N}(\frac{c_{pi}(b)}{c_i(b)}log\frac{c_{pi}(b)}{c_i(b)} + \frac{c_{pi}(b)}{c_i(b)}log\frac{c_{pi}(b)}{c_i(b)}) \quad (5)$$

$$I(y|b) = H(y) + \sum_{i=0}^{max(n(b))} \frac{c_i(b)}{N}(\frac{c_{pi}(b)}{c_i(b)}log\frac{c_{pi}(b)}{c_i(b)} + \frac{c_{pi}(b)}{c_i(b)}log\frac{c_{pi}(b)}{c_i(b)}) \quad (6)$$

The mutual information $I(y|b)$ quantifies the "amount of information" obtained about a crash through observing the presence of basic block b. In plain English, it represents the contribution of basic block b to the crash.

Mutual information with threshold. As can be seen from Equation (6), the amount of "crash relevance" is related to $max(n(b))$ and the statistics of b such as $c_i(b)$ and $c_{pi}(b)$. Its value can vary according to different $max(n(b))$. To fairly compare the amount of information for the basic blocks with different occurrence in the execution traces, we use a threshold that is automatically-determined and turn the problem into a binary classification problem (i.e., occurrence is smaller than or larger than a threshold): assume we have a threshold variable thd whose value is a non-negative integer with $thd \in [0, max(n(b)))$. Given the threshold variable, $H(y|b)$ can be the addition of $H(y|b|_{n(b)\leq thd})$ and $H(y|b|_{n(b)>thd})$, where:

$$H(y|b|_{n(b)\leq thd}) = -\frac{\sum_{i=0}^{thd} c_{pi}(b)}{\sum_{i=0}^{thd} c_i(b)}log(\frac{\sum_{i=0}^{thd} c_{pi}(b)}{\sum_{i=0}^{thd} c_i(b)}) - \frac{\sum_{i=0}^{thd} c_{fi}(b)}{\sum_{i=0}^{thd} c_i(b)}log(\frac{\sum_{i=0}^{thd} c_{fi}(b)}{\sum_{i=0}^{thd} c_i(b)}) \quad (7)$$

$$H(y|b|_{n(b)>thd}) = -\frac{\sum_{i=thd+1}^{max(n(b))} c_{pi}(b)}{\sum_{i=0}^{thd} c_i(b)}log(\frac{\sum_{i=thd+1}^{max(n(b))} c_{pi}(b)}{\sum_{i=0}^{thd} c_i(b)}) - \frac{\sum_{i=thd+1}^{max(n(b))} c_{fi}(b)}{\sum_{i=0}^{thd} c_i(b)}log(\frac{\sum_{i=thd+1}^{max(n(b))} c_{fi}(b)}{\sum_{i=0}^{thd} c_i(b)}) \quad (8)$$

Combining Equation (1), Equation (5), Equation (7) and Equation (8), we have:

$$I(y|b, thd) = H(y) - \frac{\sum_{i=0}^{thd} c_i(b)}{N}H(y|b|_{n(b)\leq thd}) - \frac{\sum_{i=thd+1}^{max(n(b))} c_i(b)}{N}H(y|b|_{n(b)>thd}) \quad (9)$$

By iterating through $[0, max(n(b)))$, we can determine the thd that maximizes $I(y|b)$:

$$\hat{thd} = \underset{thd\in[0,max(n(b))}{argmax} I(y|b, thd) \quad (10)$$

In the execution trace, when the occurrence of a basic block b is larger than $\hat{thd}$, the basic block has close relevance to label y. In Equation 10, $I(y|b, \hat{thd})$ represents the degree of "crash relevance".

Filtering of basic blocks with $y = 0$. A criterion basic block is a basic block that is closely related to the crash (i.e., when $y = 1$) and has high value of mutual information. Such a criterion basic block (or a set of criterion basic blocks) is used as the criterion in the subsequent crash grouping procedure. From the above analysis, we know that $I(y|b, \hat{thd})$ represents the contribution of basic block b to label y. However, this representation does not differentiate whether the contribution is related to normal program exit (i.e., when $y = 0$) or program crash (i.e., when $y = 1$). Namely, using the basic blocks whose $I(y|b, \hat{thd})$ is large and related to $y = 0$ would cause false negatives in crash deduplication. When the occurrence of b is larger than the threshold (i.e., $n(b) > \hat{thd}$) and the number of "normal exit" samples is larger than the number of "crashing samples", it indicates that, with such a basic block, the program is prone to exit normally. Therefore, we need to filter out the basic blocks that have a large $I(y|b, \hat{thd})$ and are closely related to normal program exit, and select criterion basic blocks from the rest. The detailed algorithm of filtering of basic blocks with y=0 is shown in Algorithm 1.

Algorithm 1 Filtering of Basic Blocks with $y = 0$

Require: $b \leftarrow$ input basic block
$\hat{thd} \leftarrow$ threshold of b
$max(n(b)) \leftarrow$ the maximum occurrence of b in the dataset
$c_{pi}(b) \leftarrow$ the amount of "normal exit" samples when occurrence of b is i
$c_{fi}(b) \leftarrow$ the amount of "crashing" samples when occurrence of b is i
$N_p \leftarrow$ the amount of "normal exit" samples in the dataset
$N_f \leftarrow$ the amount of "crashing" samples in the dataset
Ensure: whether to consider b as a criterion basic block
1: **if** $\frac{\sum_{i=\hat{thd}+1}^{max(n(b))} c_{pi}(b)}{N_p} > \frac{\sum_{i=\hat{thd}+1}^{max(n(b))} c_{fi}(b)}{N_f}$ **then**
2: **return** False
3: **else**
4: **return** True
5: **end if**

3.2 Attention Mechanism of Neural Networks

The attention mechanism was initially proposed to improve the fitting of neural networks by assigning different weights to the input sequence and minimizing the loss function [13]. In recent years, a line of research [8, 21, 24, 45] leveraged the attention mechanism for the interpretability of neural networks, allowing us to inspect the internal working of neural networks directly. The hypothesis is that the magnitude of attention weights positively correlates with how relevant a specific input region is for predicting output at each position in a sequence. It can be easily accomplished by visualizing the attention weights for a set of input and output pairs. In this paper, we borrow this idea and leverage the attention mechanism to identify the root cause of program crash by weighing each input basic block's contribution to the crash.

As discussed, the idea of attention mechanism is straightforward. For an input vector $(\vec{x}_1, \vec{x}_2, ..., \vec{x}_n)$, suppose we have:

$$\begin{aligned} \vec{v} &= \alpha_1 \vec{x}_1 + \alpha_2 \vec{x}_2 + ... + \alpha_n \vec{x}_n \\ &and\ y = f(\vec{v}), \\ where\ &\sum_i \alpha_i = 1, \alpha_i > 0. \end{aligned} \tag{11}$$

To function $y = f(\vec{x})$, α_i can be regarded as the contribution that input byte x_i makes to y, where $(\alpha_1, \alpha_2, ..., \alpha_n)$ is also known as a weighted vector. Such a function $y = f(\vec{x})$ is often utilized to determine the influence of input bytes to the output in seq2seq networks. The transition equation is as follows:

$$\begin{aligned} \vec{\alpha} &= g(\vec{x}; \vec{\theta}), \\ \vec{v} &= \alpha_1 \vec{x}_1 + \alpha_2 \vec{x}_2 + ... + \alpha_n \vec{x}_n, \\ y &= f(\vec{v}; \vec{\theta}), \\ where &\sum_i \alpha_i = 1, \alpha_i > 0 \end{aligned} \tag{12}$$

$\vec{\theta}$ is the parameter to be determined in the training process. Function $g(\vec{x}; \vec{\theta})$ is used to calculate the weight vector, which is also known as similarity function. In the dataset, $\vec{x}^i$ is the ith sample and $\vec{y}^i$ is the corresponding label. The loss function with mean square error is:

$$\begin{aligned} L(f(\vec{x}, \vec{\theta})) &= \sum_i |f(g(\vec{x}^i; \vec{\theta}) \odot \vec{x}^i; \vec{\theta}) - y^i|^2, \\ &s.t. \sum g(\vec{x}^i; \theta) = 1 \end{aligned} \tag{13}$$

However, when using the gradient descent method to minimize loss $L(f(\vec{x}, \vec{\theta}))$, it is difficult to satisfy the constraint $\sum g(\vec{x}^i; \theta) = 1$ and get $\vec{\theta}$. Therefore, *softmax* function is adopted as the activation function of $g(\vec{x}, \vec{\theta})$ in the design of networks, given that the sum of *softmax* function's output equals to 1. The transition equation with *softmax* becomes:

$$\begin{aligned} \vec{\alpha} &= softmax(g(\vec{x}; \vec{\theta})), \\ \vec{v} &= \alpha_1 \vec{x}_1 + \alpha_2 \vec{x}_2 + ... + \alpha_n \vec{x}_n, \\ y &= f(\vec{v}; \vec{\theta}), \\ softmax(x_i) &= \frac{e^{x_i}}{\sum_j e^{x_j}} \end{aligned} \tag{14}$$

And the loss function becomes:

$$L(\vec{\theta}) = \sum_i |f(softmax(g(\vec{x}^i; \vec{\theta})) \odot \vec{x}^i; \vec{\theta})) - y^i|^2 \tag{15}$$

The network that we designed (as described in Section 4.3) follows the above transition equation. In fact, Equation (14) is the core architecture of the attention mechanism and such an architecture can be used to determine the relevance of the input bytes and the output. In particular, under this architecture, we are able to get the $\vec{\theta}$ by minimizing $L(f(\vec{x}, \vec{\theta}))$ with the gradient descent. Function $g(\vec{x}; \vec{\theta})$ or $f(\vec{x}; \vec{\theta})$ could be convolutional neural networks (CNN), recurrent neural networks (RNN) or fully connected networks. While in seq2seq networks, $g(\vec{x}; \vec{\theta})$ is LSTM and $f(\vec{x}; \vec{\theta})$ is a fully connected network.

4 DESIGN OF DEFAULT

The high-level design of our system is illustrated in Figure 1. The input is a set of crashing execution traces produced by fuzzers. Those traces are fed into the crash deduplication module, which buckets crashing traces into multiple categories according to their root cause and outputs one representative crashing trace with a unique root cause from each category. Then the representative crashing trace of each category is sent to the filtering module, which consists of two filtering steps – function filtering and basic block filtering. It filters out the vast majority of functions and basic blocks in the trace that are irrelevant to the crash, so that the neural network in the subsequent module can take the shortened traces as its inputs. In the end, the fault localization module leverages the attention mechanism to identify a set of basic blocks that contribute to the crash.

4.1 Crash Deduplication

There are two steps for crash deduplication: first, we group crashing traces into multiple groups based on the calculated mutual information of basic blocks; then, we select a representative crashing trace from each group. The root cause of the selected representative crashes is different from each other[2].

Grouping. Assume that the root cause of crashing trace T_A includes basic block b_A, and b_A is not included in crashing trace T_B. Thus, we can put those two traces into two groups: one group contains traces with b_A, and the other contains traces without b_A. Motivated by that, we separate the dataset into two groups: Group G_A is the set of crashing traces with basic block $\hat{b}$; the other group G_B the set of crashing traces without basic block $\hat{b}$. The basic block $\hat{b}$ has the highest $I(y|b, \hat{thd})$ in the dataset D, which is believed to have a significant contribution to a specific crash.

We repeat the above grouping step for group G_B until there is no crashing trace left. After such a preliminary grouping process, most of the crashing traces in a group have the same root cause.

Since the traces with different basic block statistics could be categorized into different groups, such a grouping algorithm gives false positives. Nevertheless, it gives no false negatives. For instance, assume that basic block b_c in trace T_C and basic block b_d in trace T_D are relevant to the same root cause (namely, trace T_C and trace T_D share the same root cause), but b_c and b_d have different occurrences. This happens when the root cause leads to different crashing points in the program. As a result, our algorithm would put trace T_C and trace T_D into two different groups. Fortunately, although more groups are produced than the ground truth, the over-categorization would not miss any root cause of the crashing traces produced by fuzzers. We show the complete grouping algorithm in Algorithm 2.

Selecting representative crashing traces. Within a group, when a crashing trace contains a criterion basic block $\hat{b}$ and another basic block b_1 which is the root cause of another unique crash, this crashing trace cannot represent this group. It would lead to false positives and false negatives in subsequent steps. Therefore, to represent this group, we need to select a crashing trace, whose root-cause basic blocks only include a criterion basic block $\hat{b}$, without

[2]Our approach does not guarantee that the crash de-duplication module gives unique crashes. We only found that there are no false negatives in the evaluation.

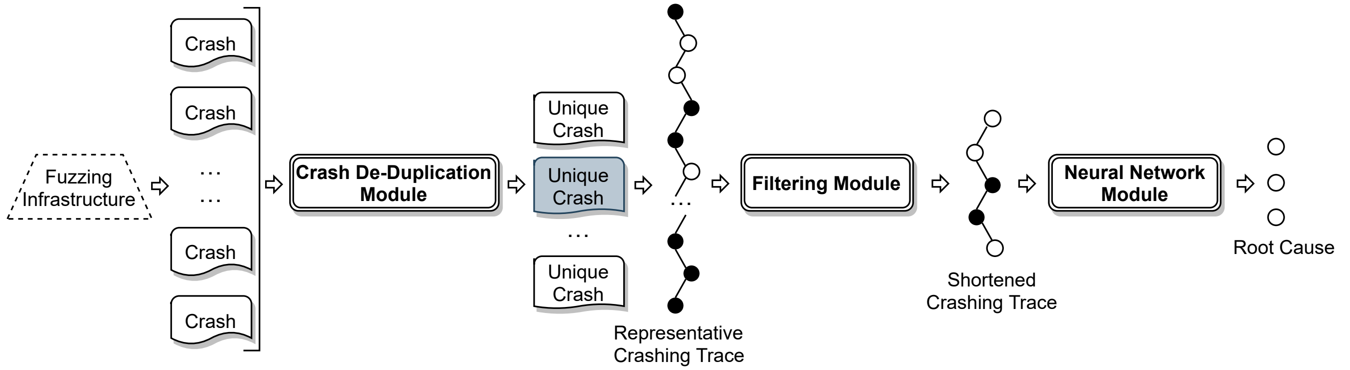


Figure 1: Overall design of DeFault.

Algorithm 2 Mutual Information-based Grouping

Require: $\mathbb{B}_f \leftarrow$ set of crashing traces **B**, $\mathbb{B}_p \leftarrow$ set of non-crashing traces **B**
Ensure: **CC** $\leftarrow$ set of grouped crashing traces, **criterion** $\leftarrow$ set of criterion basic block of each group
1: **CC** $\leftarrow$ [ ]
2: **criterion** $\leftarrow$ [ ]
3: **D** $\leftarrow (<\mathbb{B}_f, 1>) \cup (<\mathbb{B}_p, 0>)$
4: calculate mutual information for each basic block in **D**
5: select b that is related to $y = 1$ and has the largest mutual information value
6: **if** $\sum_i^{\hat{thd}} c_{fi}(b) == 0$ **then**
7: CC.$append(B_f)$
8: **criterion**.$append(b)$
9: **return** CC, **criterion**
10: **else**
11: $new\mathbb{B}_f \leftarrow \{\mathbf{B}|\mathbf{B} \in \mathbb{B}_f, b \in \mathbf{B}, n(b) > \hat{thd}\}$
12: CC.$append(\mathbb{B}_f)$
13: **criterion**.$append(b)$
14: $\mathbb{B}_f \leftarrow \{\mathbf{B}|\mathbf{B} \notin new\mathbb{B}_f, \mathbf{B} \in \mathbb{B}_f\}$
15: goto line 3
16: **end if**

Algorithm 3 Selection of Representative Crashing Trace

Require: CC $\leftarrow$ groups of crashing traces, **Criterion** $\leftarrow$ the set of criterion basic blocks
Ensure: **Unique** $\leftarrow$ the set of crashing traces after selection
1: **Unique** $\leftarrow$ [ ]
2: $Score \leftarrow$ score dictionary
3: **for** $B \in$ CC **do**
4: **for** $b \in B$ **do**
5: **for** $t \in$ **Criterion do**
6: **if** $t \in b$ && $n(t) > \hat{thd}$ **then**
7: $Score[b] \leftarrow Score[b] + 1$
8: **end if**
9: **end for**
10: **end for**
11: $b \leftarrow min(Score[\{b|b \in B\}])$
12: **Unique**.$append(b)$
13: **end for**
14: **return** **Unique**

the basic blocks of other unique crashes. To this aim, we rank the crashing traces within a group, according to the occurrence of all criterion basic blocks. A crashing trace obtains higher score if it has more occurrences of criteria basic blocks. In the end, we select the crashing trace with the least score to represent its group, meaning that the crashing trace is more "pure". The algorithm is described in Algorithm 3.

4.2 Filtering

Once the representative crashing trace is selected, the filtering module (shown in Figure 2) performs preliminary filtering to filter out functions and basic blocks irrelevant to the crash. It can vastly shorten the length of crashing traces and facilitate the neural network module to process the shortened traces.

Constructing Datasets. To cover more basic blocks and comprehensively assess how the executions of different basic blocks affect a crash, we construct a dataset that is bred from a unique crashing trace. We use afl-fuzz to mutate a single crashing input and breed inputs that explore different basic blocks. The output of afl-fuzz can be categorized into crashing inputs and non-crashing inputs. Note that in extreme cases, for example, when afl-fuzz triggers new crashes, the follow-up procedures would not be affected. The non-crashing inputs are obtained by randomly mutating the crashing inputs. In this way, a large portion of the basic blocks in crashing traces would also appear in the non-crashing trace, and those basic blocks are not related to the crash. Consequently, in the dataset, the statistics of those overlapped basic blocks are significantly different from the statistics of the crash-related basic blocks. It helps to form a dataset with differentiable and adequate samples. Note that this dataset also serves for the training of the neural network in Section 4.3.

Filtering out irrelevant functions. In the program, there exist some low-level functions like data copy functions, operation functions of linked lists, and constructors/destructors of structs or objects. When those functions are invoked near the root cause, the mutual information of those functions' basic blocks would be very close to the mutual information of the actual root cause, which brings false positives to the results. Therefore, we calculate mutual

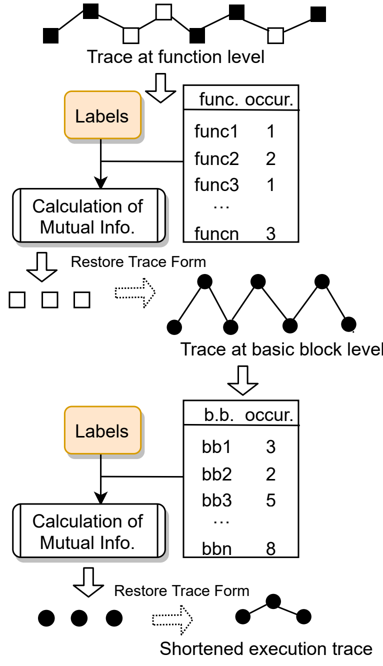


Figure 2: Illustration of filtering.

information at the function level to filter out the irrelevant functions to the actual root cause.

The dataset is composed of execution traces at function level with their corresponding labels: $\mathbb{D} =< \mathbb{F}, \mathbb{Y} >$. In the dataset, F_i is the ith trace of $\mathbb{F}$ where $F = (f_1, f_2, ..., f_{N_f})$ is a sequence of functions in that trace and y_i represents whether the execution trace corresponds to a crash. We calculate the mutual information of function f_i to label y, to represent f_i's contribution to label y:

$$I(y|f_j, \hat{thd}) = H(y) - \frac{\sum_{i=0}^{thd} c_i(f_j)}{N} H(y|f_j|_{n(f_j) \leq \hat{thd}}) - \frac{\sum_{i=thd+1}^{max(n(f_j))} c_i(f_j)}{N} H(y|f_j|_{n(f_j) > \hat{thd}}) \tag{16}$$

where N is the amount of samples in the dataset, $H(Y)$ is the entropy of label y, $c_i(f_j)$ is the amount of samples where f_j's occurrence is i, $H(y|x)$ is the conditional entropy, and $\hat{thd}$ is:

$$\hat{thd} = \underset{thd \in [0, max(n(f_j))}{argmax} I(y|f_j, thd) \tag{17}$$

After filtering, function f_i's contribution to label y can be regarded as $r = I(y|f_i, \hat{thd})/H(y)$ with $r \in (0, 1]$, where $I(y|f_j, \hat{thd})$ is the mutual information and $H(y)$ is the entropy of label y. When $r = 1$, we can infer that an execution trace is a crashing trace by observing that f_i appears in the trace. Similarly, when r approaches 0, the (non-)existence of f_i has little impact on whether an execution trace is a crashing trace or not. Therefore, we filter out the functions whose r is smaller than 0.5 [3] and obtain the traces with selected functions:

$$F_s = \{f | f \in F, \frac{I(y|f, \hat{thd})}{H(y)} > 0.5\} \tag{18}$$

Filtering out irrelevant basic blocks. After the filtering of irrelevant functions, the next step is to filter out irrelevant basic blocks. We denote the dataset as $\mathbb{D} =< \mathbb{B}, \mathbb{Y} >$, where n is the amount of samples, $\mathbb{B} = \{\mathbf{B}_1, \mathbf{B}_2, ..., \mathbf{B}_n\}$ is a set of execution traces at basic block level, and $\mathbb{Y} = \{y_1, y_2, ..., y_n\}$ is the set of labels for the execution traces. After the filtering of irrelevant functions in the dataset, the filtered execution trace becomes $B_r = (b_{r_1}, b_{r_2}, ..., b_{r_n})$, where b_{r_i} is a basic block. By calculating the mutual information of b_{r_j} to label $y = 0$, we can select the set of basic blocks that contribute more to the crash. The mutual information of basic block b_{r_j} to label y is given by:

$$I(y|b_{r_j}, \hat{thd}) = H(y) - \frac{\sum_{i=0}^{thd} c_i(b_{r_j})}{N} H(y|b_{r_j}|_{n(b_{r_j}) \leq \hat{thd}}) - \frac{\sum_{i=thd+1}^{max(n(b_{r_j}))} c_i(b_{r_j})}{N} H(y|b_{r_j}|_{n(b_{r_j}) > \hat{thd}}) \tag{19}$$

where $\hat{thd}$ is:

$$\hat{thd} = \underset{thd \in [0, max(n(b_{r_j}))}{argmax} I(y|b_{r_j}, thd) \tag{20}$$

We then filter out the basic block that are closely related to label $y = 0$. Similar to function filtering, we calculate the ratio $r = I(y|b_{r_j}, \hat{thd})/H(y)$ and filter out basic block whose r is less than 0.8. The r value is set higher than the r of function filtering because the selection on basic blocks is more fine-grained. In the end, we obtain the trace with selected basic blocks: $B_v = (b_{v_1}, b_{v_2}, ..., b_{v_{N_v}})$ where:

$$b_v = \{b | b \in B_r, \frac{I(y|b_{r_j}, \hat{thd})}{H(Y)} > 0.8\} \tag{21}$$

4.3 Fault Localization

After filtering irrelevant functions and basic blocks, the size of traces has been largely reduced, which becomes suitable to feed neural networks. In the fault localization module, we leverage the neural network to identify the root cause's basic blocks. The basic blocks in the short traces have a high value of mutual information to label $y = 1$. However, so far, they still cannot be regarded as the root cause because some basic blocks that are not the root cause but near the root cause also get high mutual information values. The fundamental cause is that basic blocks' statistical information does not contain sequence information in its execution trace. With all existing approaches fail to recover the sequence information in execution traces, especially when handling the dependence in long sequences, we leverage LSTM with the attention mechanism to model and capture the sequence information about the root cause. More specifically, we have the dataset $\mathbb{D} =< \mathbb{B}_v, \mathbb{Y} >$, where $\mathbb{B}_v$ is the set of execution traces composed of sequences of selected basic blocks and $\mathbb{Y}$ is the set of corresponding labels. This module utilizes the neural network to calculate the relevance score of each basic block to the crash. The root cause is the basic blocks that contribute to the crash, which is indicated by the relevance score (i.e., the

[3] The filtering threshold r is empirically set to 0.5. We found that the following modules produced satisfactory results with this value.

higher relevance score of a basic block indicates more contribution to the crash).

Input and output of neural network. The data used for training the network are the shortened basic blocks. The positive samples are the shortened crashing traces and negative samples are the shortened non-crashing traces. We use one-hot vectors to encode the input: denoting the number of basic block b_{v_i} in trace B_v as n_b. Then basic block $b_{v_i} \in B_v$ is represented as $\vec{x}_i$ with $\vec{x}_i = \{0, 1\}^{n_b}$ and $\sum_{j=0}^{n_b} |x_{ij}|^2 = 1$. The input is $\vec{X} = (\vec{x}_1, \vec{x}_2, ..., \vec{x}_n)$. Since the length of the input varies, we choose the longest B_v as the input length and pad the short inputs with zero vectors whose magnitude is n_b. Every basic block is independent with each other in the input. The sequence information about root cause among basic blocks needs to be recovered through weight assignment of the neural network. The output of neural network is the boolean value that represents whether a crash is triggered or not. Besides, we use the up-sampling [4] to balance the negative samples and positive samples in the dataset.

Network structure. The network structure is the classic LSTM network with attention mechanism. After one-hot encoding, we send execution trace B_v to the LSTM network, the output of the LSTM network is: $o_i^{LSTM} = f_{LSTM}(\vec{x}_i, o_{i-1}^{LSTM})$. In the output of the LSTM network, the ith element is related to all the previous $i - 1$ elements. We then send $\vec{o}^{LSTM}$ to the $softmax$ layer, and get:

$$\alpha_i = \frac{e^{o_i^{LSTM}}}{\sum_{j=0}^{n} e^{o_j^{LSTM}}} \tag{22}$$

In the end, we multiply vectors $\vec{\alpha}$ and $\vec{X}$ in the fully-connected layers and get the final output:

$$\begin{gathered} output = f_{dense}(\vec{v}) \\ \vec{v} = \alpha_1 * \vec{x}_1 + \alpha_2 * \vec{x}_2 + ... + \alpha_n * \vec{x}_n, \end{gathered} \tag{23}$$

In the $softmax$ layer, we have $\sum_i \alpha_i = 1$ and $\alpha_i > 0$. Also ,the output is of the form $y = f(\alpha_1 * x_1 + \alpha_2 * x_2 + ... + \alpha_n * x_n)$. Thus, $\vec{\alpha}$ is the relevance score vector (i.e., the weight vector), and α_{x_i} represents the contribution of x_i to the output y.

Calculation of relevance score. After network training, we get the relevance score vector $\vec{\alpha}^i$ for positive sample B_{vp_i} in the dataset. The relevance score of basic block b_{v_j} in B_{vp_i} is $r_j = \sum_i \alpha_{v_j}^i$. For all the basic blocks in the positive samples, we need to categorize them into two groups: one group includes basic blocks that are believed to be the root cause; the other group includes the rest of the basic blocks. The criterion for categorization is that, in each group, the variance of relevance scores of basic blocks is the smallest. The detailed algorithm is described in Algorithm 4.

Algorithm 4 Selection of Root-Cause Basic Blocks

Require: $r_j \leftarrow$ relevance score of basic block b_{v_j}
  $n \leftarrow$ amount of basic blocks
  $\mu \leftarrow$ variance
Ensure: $score \leftarrow$ threshold of relevance score

```
1: sort(r⃗)
2: min ← μ(r_0) + μ(r_1, .., r_{n−1})
3: score ← r_0
4: for i ∈ [0,n) do
5:     tmp ← μ(r_0, .., r_i) + μ(r_{i+1}, .., r_{n−1})
6:     if tmp < min then
7:         min ← tmp
8:         score ← r_j
9:     end if
10: end for
```

5 EVALUATION

We implemented a full-featured prototype of DeFault. To evaluate its efficacy and performance, we conducted comprehensive real-world experiments. In this section, we present and discuss the corresponding experiment results.

5.1 Experimental Setting

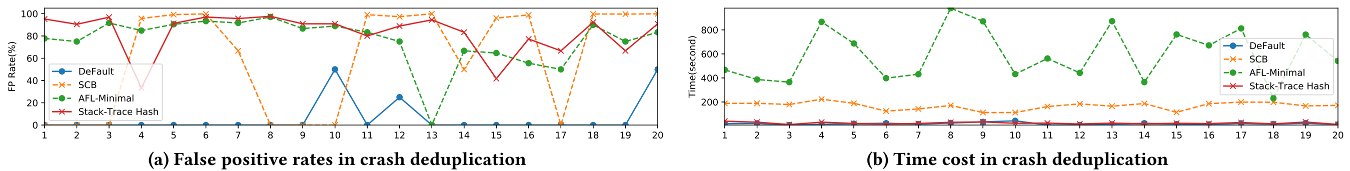
Target programs and testing environment. We evaluated our tool with 20 programs, including 8 CGC programs and 12 real-world programs. To reasonably choose the programs, we select the programs that have known crashes[4] and belong to different software categories from the CGC program repositories and the CVE list, without examining the details of crash and program internals. The functionalities of the programs include image processing, document parsing, compilation, and audio processing. The experiments run in Ubuntu 18.04, with Intel i9 7900X, 48GB DDR4, and RTX 2080Ti (11GB VRAM). The version of AFL is 2.52b. TensorFlow and Keras are used for neural network training.

Triggering crashes. The crashing inputs are produced with afl-fuzz. With sufficient fuzzing time (one week), afl-fuzz reproduced crashes in all 8 CGC programs that we selected. However, we also selected 16 real-world programs, and afl-fuzz only reproduced crashes in 12 of them (with some initial seed inputs). On average, each program contains 2.15 faults. After fuzzing, the average amount of crashes is 516.6 for each program fault.

Neural network setup. The parameter of the neural network, LSTM (256 units) with attention mechanism. The mean square error is used as the loss function, and the Adam optimizer is used with an initial learning rate 10^{-6}. We stop the training when the fitting rate becomes 99% or when the iteration round reaches 100.

Ground truth. To obtain the ground truth, we manually inspect each crashing trace backward from the crashing point to the root cause and record the crashing point (one single basic block) and the root cause (multiple basic blocks), by debugging and reverse engineering. Specifically, For crash de-duplication, we used scripts to determine whether a crashing input would satisfy the constraints (that we manually written after examining the root cause) to trigger the bug. For fault localization, we manually determined the root-cause basic blocks after manual inspection. The root causes of CGC programs are publicly available. For the real-world buggy software, we also refer to the bug report from the CVE reference. If the root cause of two different crashing traces is equivalent, we categorize them into the same group, no matter how diverse are their crashing points.

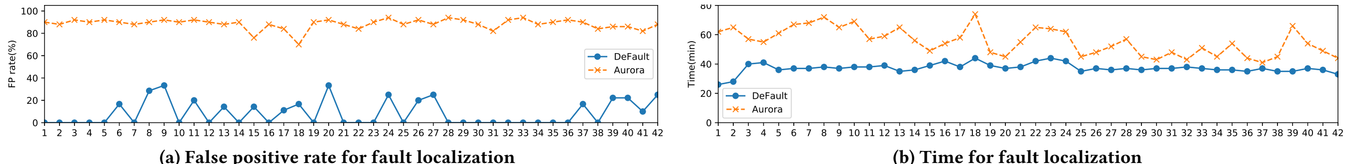
[4]For real-world programs, the crashes have been fixed in new versions.



(a) False positive rates in crash deduplication

(b) Time cost in crash deduplication

Figure 3: Accuracy and performance of crash deduplication.



(a) False positive rate for fault localization

(b) Time for fault localization

Figure 4: Accuracy and performance of fault localization.

5.2 Crash Deduplication: Efficacy and Accuracy

Comparison with existing tools. Regarding crash deduplication, we compared our approach with *Semantic Crash Bucketing* (SCB for short) [37], afl-fuzz's deduplicator [49], and Honggfuzz's deduplicator [2]. In particular, SCB achieves crash deduplication by automatically fixing bugs, which identifies crashes belonging to the same bug. This approach is relatively accurate but only targets two types of bugs, namely buffer overflows and null pointer dereference, which is less applicable to general bugs. Afl-fuzz's deduplicator is commonly used in industry. It first reduces the crashing trace using afl-tmin [1] and then calculates the hash of the crashing path. Crash bucketing is achieved by comparing the similarity of the hashes. On the contrary, Honggfuzz's deduplicator compares the similarity of call stack hashes, working at function call level. In the experiment, since afl-tmin's reduction on a single crashing trace introduces time overhead, we run afl-tmin in parallel with 20 cores.

False negatives. As can be seen from Table 1, there is no false negative given by DeFault. Neither SCB nor afl-fuzz's deduplicator produces any false negatives. However, after investigating Honggfuzz's deduplication results, we still found one case of false negative: in the software `listswf`, when multiples root causes exist in the same function, their execution traces are the same at the function level. Given that Honggfuzz's deduplication categorizes crashes based on the execution traces at the function call level, it is unable to identify and differentiate two different kinds of the root cause that occur in the same function and therefore misses some crashes that have been triggered by the fuzzer.

False positives. We show the false positive rate of all the tools in Figure 3a. Overall, DeFault outperforms all three tools by giving less false positives on 19 cases. SCB's high false positive rate can be attribute to its limited support of bugs, namely buffer overflows and null pointer dereference. For Afl-fuzz's deduplication, given that afl-tmin's reduction cannot completely remove redundant paths and one single bug can be triggered from multiple paths, afl-fuzz also gives high false positives. Similarly, when a bug can be triggered from multiple paths, the call stack hashes are also different. Therefore, Honggfuzz's deduplicator also gives false positives.

Particularly, for case 18 and case 20 where DeFault gives false positives, the crashes are caused by use-after-free vulnerabilities. The root cause of the vulnerabilities is typically the misuse of `free()` operation, while the crashing point could be large variant based on different runtime memory layouts. Since the use-after-free vulnerabilities are triggered but do not cause any crashes in some positive samples, the basic blocks of root cause also appear in positive samples, which leads to low mutual information values and causes false positives.

5.3 Crash Deduplication: Performance

The number of crashing trace for each program is 1110 on average. Processing 1110 traces takes DeFault 19 seconds. With the same workload, SCB, afl-fuzz and Honggfuzz takes 168 seconds, 595 seconds and 24 seconds, respectively. The time cost of DeFault is only 11.3%, 3.1%, and 79% of that of SCB, afl-fuzz and Honggfuzz, respectively.

The comparison of DeFault and other three tools's time cost is shown in Figure 3b. As can be seen, to process execution traces, afl-fuzz's deduplicator takes much more time than DeFault. The main reason is that afl-fuzz's deduplication is based on the comparison among execution traces. The time cost increases exponentially as the amount of the traces grows, because each trace needs to be compared with all other traces. On the contrary, DeFault analyzes the statistics of basic blocks, and the computation of mutual information values only involves some of the basic blocks. The values of mutual information are stored and indexed from hash tables. As such, it takes much less time for DeFault.

5.4 Fault Localization: Efficacy and Accuracy

Assessment of effectiveness. Traditional program spectrum-based fault localization relies on the EXAM curve [17] to assess effectiveness. A point (x, y) in the EXAM curve represents the suspicious score of a basic block. However, such measurement

Table 1: Overall Results

ID	Program	Unique Crash by AFL (#)	DeFault								SCB / afl-fuzz / Honggfuzz									Aurora		
			Crash Deduplication				Fault Localization				Crash Deduplication									Fault Localization		
			Time (s)	Groups (#)	Ground Truth	F.P. F.N.	Time (s)	Basic Blocks (#)	Ground Truth	F.P. F.N.	Time (s) SCB	Time (s) afl-fuzz	Time (s) Honggfuzz	# of Groups SCB	# of Groups afl-fuzz	# of Groups Honggfuzz	F.P. F.N. SCB	F.P. F.N. afl-fuzz	F.P. F.N. Honggfuzz	Time (s)	Basic Blocks (#)	F.P. F.N.
1	cflow	441	20	2	2	0/0	1560	5	5	0/0	188	467	40	2	9	42	0/0	7/0	40/0	3720	50	45/0
							1680	6	6	0/0										3900	50	44/0
2	mp3again	329	22	2	2	0/0	2400	4	4	0/0	189	387	32	2	8	21	0/0	6/0	19/0	3420	50	46/0
							2460	5	5	0/0										3300	50	45/0
3	jhead	408	8	1	1	0/0	2160	4	4	0/0	179	365	12	1	12	32	0/0	11/0	31/0	3660	50	46/0
4	listswf	1572	14	5	5	0/0	2220	6	5	1/0	223	867	32	117	33	12	112/0	28/0	4/3	4020	50	45/0
							2220	6	6	0/0										4080	50	44/0
							2280	7	5	2/0										4320	50	45/0
							2220	6	4	2/0										3900	50	46/0
							2280	5	5	0/0										4140	50	45/0
5	GraphicsMagick	760	19	2	2	0/0	2280	5	4	1/0	189	687	21	243	21	23	241/0	19/0	21/0	3420	50	46/0
							2340	5	5	0/0										3540	50	45/0
6	jasper	479	23	1	1	0/0	2100	7	6	1/0	124	398	16	479	15	33	478/0	14/0	32/0	3900	50	44/0
7	pdftopng(xpdf)	981	16	1	1	0/0	2160	5	5	0/0	142	431	21	3	12	23	2/0	11/0	22/0	3360	50	45/0
8	nasm	3713	25	1	1	0/0	2340	14	12	2/0	171	981	31	1	34	42	0/0	33/0	41/0	2940	50	38/0
9	latex2rtf	1787	34	2	2	0/0	2520	6	6	0/0	112	871	34	2	15	22	0/0	13/0	20/0	3240	50	44/0
							2280	9	8	1/0										3480	50	42/0
10	mruby	887	43	2	1	1/0	2640	18	15	3/0	112	432	23	1	9	11	0/0	8/0	10/0	4440	50	35/0
11	tiffcp(libtiff)	1092	9	3	3	0/0	2340	6	5	0/0	162	562	24	342	18	15	339/0	15/0	12/0	2880	50	45/0
							2220	6	4	2/0										2700	50	46/0
							2280	6	6	0/0										3300	50	44/0
12	pdfrescurrent	682	8	4	3	1/0	2520	8	8	0/0	184	442	17	112	12	27	109/0	9/0	24/0	3900	50	42/0
							2640	5	5	0/0										3840	50	45/0
							2520	4	3	1/0										3720	50	47/0
13	FileSys	1231	13	1	1	0/0	2100	6	6	0/0	165	872	24	1231	1	18	1230/0	0/0	17/0	2700	50	44/0
14	Street map service	762	23	2	2	0/0	2220	5	4	1/0	187	365	18	4	3	12	2/0	2/0	10/0	2880	50	46/0
							2160	8	6	2/0										3120	50	44/0
15	Kaprica Script Interpreter	1208	15	6	6	0/0	2220	3	3	0/0	114	762	23	152	17	12	146/0	11/0	5/1	3420	50	47/0
							2160	4	4	0/0										2700	50	46/0
							2220	6	6	0/0										2580	50	44/0
							2220	9	9	0/0										2880	50	41/0
							2280	4	4	0/0										2580	50	46/0
							2220	3	3	0/0										3060	50	47/0
16	simple integer calculator	1082	13	4	4	0/0	2160	6	6	0/0	186	671	20	365	9	22	361/0	5/0	17/0	2700	50	44/0
							2160	5	5	0/0										3240	50	45/0
							2100	4	4	0/0										2640	50	46/0
							2220	6	5	1/0										2460	50	45/0
17	CGCRPC_Server	1876	22	1	1	0/0	2100	8	8	0/0	199	812	29	1	2	3	0/0	1/0	2/0	2700	50	42/0
18	Shortest Path Tree Calculator	365	16	1	1	0/0	2100	9	7	2/0	198	231	18	365	10	13	364/0	9/0	12/0	3960	50	43/0
19	SOLFEDGE	1763	24	2	2	0/0	2220	9	7	2/0	167	761	33	564	8	6	562/0	6/0	4/0	3240	50	43/0
							2160	10	9	1/0										2940	50	41/0
20	User_Manager	798	13	2	2	0/0	1980	8	6	2/0	172	541	13	798	6	11	797/0	5/0	10/0	2640	50	44/0

does not make sense in practice, as analysts would wish a tool to output exact results rather than the top n suspiciousness rank of the basic blocks. For example, when the root cause is ranked 9 and 10 within the top 10 suspicious basic blocks, analyzing the first 8 basic blocks does not help. As such, in our evaluation, we use accuracy and false positive rate as indicators. For a given binary program and a crashing input, the number of false positives is $F = Num(O) - Num(O \cap A)$, where O is the set of basic blocks reported by DeFault and A is the set of basic blocks of the root cause. The accuracy and the false positive rate are defined as $fp = \frac{F}{A} \times 100\%$ and $fn = \frac{F}{Num(O)} \times 100\%$, respectively.

False positives. We compare DeFault with *Aurora* [10], a recent work that is based on analyzing the statistics of execution paths. *Aurora* outputs top 50 basic blocks as its results of root cause identification. On the whole, DeFault demonstrates relatively low false positive rates in fault localization: the average false positive rate on 42 crash cases is 9.2%. Note that no false negative occurs in both DeFault and *Aurora*. Figure 4a shows the false positive rate on 42 crash cases in detail. The average number of basic blocks that are reported as false positives is 0.7. The presence of use-after-free vulnerabilities causes false positives, but no crash is triggered. Namely, in the presence of a use-after-free vulnerability, when there is no memory operation on the piece of memory that is freed, no crash would be triggered. The behaviors of the root cause and that of the crash point are the same, which causes false positives on the reported results.

5.5 Fault Localization: Performance

On average, the time cost of DeFault to locate each root cause is 37 minutes. For the mutual-information-based filtering, the time consumption is highly related to the scale of execution trace, including the length of execution trace and the amount of different basic blocks. A longer execution trace with a large amount of different basic blocks takes more time to process. For the neural network module, the time cost is related to the size of the training dataset, network input size, and the number of parameters. With

the attention mechanism and the high-performance GPU, training time and localization time are controlled within 6.6 minutes on average. By that, time is mostly spent on recording execution traces. Figure 4b shows the comparison of time cost for DeFault and *Aurora*.

6 DISCUSSION

The proposed approach locates root cause at control flow level and relies on the sequence information of execution traces and whether the dataset contains rich samples. As such, our approach is less effective when execution trace is relatively short and when triggering the vulnerability depends on data flows. Fortunately, the root cause of such vulnerabilities can be effectively identified with taint analysis [52]. On the other hand, DeFault is not applicable when triggering the vulnerabilities requires to solve complicated constraints on their paths, especially for the programs with cryptographic algorithms or checksum functions. The branches with complicated constraints cannot be easily reached with the fuzzer's simple mutation strategies. This results in the fact that the inputs are less explosive, which affects the accuracy of the neural network's fitting. To this end, as long as more diverse and adequate positive samples are provided (more paths are explored), the network can assign accurate weights to each basic block. Therefore, to improve the fitting accuracy, the trade-off is to spend time and sufficiently mutate and produce samples to train the network.

7 CONCLUSION

In this paper, we have presented DeFault, an end-to-end solution for crash triage of general programs. The core insight of our solution is to leverage mutual information of basic blocks to represent "crash relevance". Implementing the insights also involves a set of new algorithms. In the evaluation, we compared DeFault with state-of-the-art solutions, which demonstrates considerable time efficiency and accuracy in both crash de-duplication and fault localization.

8 ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their insightful comments. Wenrui Diao was partially supported by National Natural Science Foundation of China (Grant No. 61902148) and Qilu Young Scholar Program of Shandong University.

REFERENCES

[1] "american fuzzy lop - test case minimizer," https://github.com/google/AFL/blob/master/afl-tmin.c, Accessed: Sep 2020.
[2] "Honggfuzz," https://github.com/google/honggfuzz, Accessed: Sep 2020.
[3] "OSS-Fuzz: Continuous Fuzzing for Open Source Software," https://github.com/google/oss-fuzz, Accessed: Sep 2020.
[4] "Up-sampling," https://en.wikipedia.org/wiki/Upsampling, Accessed: Sep 2020.
[5] R. Abreu, P. Zoeteweij, and A. J. C. V. Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques-mutation*, 2007.
[6] H. Agrawal and J. R. Horgan, *Dynamic program slicing*, 1990.
[7] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, "Fault localization using execution slices and dataflow tests," in *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95*, 2002.
[8] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.
[9] K. Bartz, J. W. Stokes, J. C. Platt, R. Kivett, and G. Loihle, "Finding similar failures using callstack similarity," in *Third Workshop on Tackling Computer Systems Problems with Machine Learning Techniques, SysML 2008, December 11, 2008, San Diego, CA, USA, Proceedings*, 2008.
[10] T. Blazytko, M. Schlögel, C. Aschermann, A. Abbasi, J. Frank, S. Wörner, and T. Holz, "AURORA: Statistical Crash Analysis for Automated Root Cause Explanation," *USENIX Security*, 2020. [Online]. Available: https://github.com/RUB-SysSec/
[11] C. Cadar, D. Dunbar, and D. Engler, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI'08 Proceedings of the 8th USENIX conference on Operating systems design and implementation*, 2008, pp. 209–224.
[12] S. Chandra, E. Torlak, S. Barman, and R. Bodik, "Angelic debugging," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 121–130.
[13] S. Chaudhari, G. Polatkan, R. Ramanath, and V. Mithal, "An attentive survey of attention models," *arXiv preprint arXiv:1904.02874*, 2019.
[14] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: a platform for in-vivo multi-path analysis of software systems," in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, vol. 46, no. 3, 2011, pp. 265–278.
[15] J. S. Collofello and L. Cousins, "Towards automatic software fault location through decision-to-decision path analysis," *afips*, 1899.
[16] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel, "Rebucket: a method for clustering duplicate crash reports based on call stack similarity," in *Proceedings of the 34th International Conference on Software Engineering*, 2012.
[17] H. A. de Souza, M. L. Chaim, and F. Kon, "Spectrum-based Software Fault Localization: A Survey of Techniques, Advances, and Challenges," pp. 1–46, 2016. [Online]. Available: http://arxiv.org/abs/1607.04347
[18] T. Dhaliwal, F. Khomh, and Y. Zou, "Classifying field crash reports for fixing bugs: A case study of mozilla firefox," in *IEEE International Conference on Software Maintenance*, 2011.
[19] D. Hao, Y. Pan, L. Zhang, W. Zhao, H. Mei, and J. Sun, "A similarity-aware approach to testing based fault localization," 2005, pp. 291–294.
[20] D. Hao, L. Zhang, H. Zhong, H. Mei, and J. Sun, "Eliminating harmful redundancy for testing-based fault localization using test suite reduction: an experimental study," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*, 2005, pp. 683–686.
[21] X. He, Z. He, J. Song, Z. Liu, Y.-G. Jiang, and T.-S. Chua, "Nais: Neural attentive item similarity model for recommendation," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 12, pp. 2354–2366, 2018.
[22] D. Kim, Y. Tao, S. Kim, and A. Zeller, "Where should we fix this bug? a two-phase recommendation model," *IEEE Transactions on Software Engineering*, vol. 39, no. 11, pp. 1597–1610, 2013.
[23] S. Kim, T. Zimmermann, and N. Nagappan, "Crash graphs: An aggregated view of multiple crashes to improve crash triage," in *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks (DSN)*, 2011, pp. 486–493.
[24] J. Li, W. Monroe, and D. Jurafsky, "Understanding neural networks through representation erasure," *arXiv preprint arXiv:1612.08220*, 2016.
[25] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," *Acm Sigplan Notices*, 2005.
[26] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical debugging: A hypothesis testing-based approach," *IEEE Transactions on Software Engineering*, vol. 32, no. 10, pp. 831–848, 2006.
[27] C. Liu and J. Han, "Failure proximity: a fault localization-based approach," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, 2006, pp. 46–56.
[28] F. Long, S. Sidiroglou-Douskos, and M. Rinard, "Automatic runtime error repair and containment via recovery shepherding," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, vol. 49, no. 6, 2014, pp. 227–238.
[29] N. Modani, R. Gupta, G. Lohman, T. Syeda-Mahmood, and L. Mignet, "Automatically identifying known software problems," in *2007 IEEE 23rd International Conference on Data Engineering Workshop*, 2007, pp. 433–441.
[30] S. S. Murtaza, M. Gittens, and N. Madhavji, "Discoveringthe fault origin from field traces," in *International Symposium on Software Reliability Engineering*, 2008.
[31] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *Acm Transactions on Software Engineering & Methodology*, vol. 20, no. 3, pp. 1–32, 2011.
[32] S. Nessa, M. Abedin, W. E. Wong, L. Khan, and Y. Qi, "Software fault localization using n-gram analysis," 2008, pp. 548–559.
[33] V. T. Pham, S. Khurana, S. Roy, and A. Roychoudhury, "Bucketing failing tests via symbolic analysis," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10202 LNCS, pp. 43–59, 2017.
[34] A. Podelski, M. Schäf, and T. Wies, "Classifying bugs with interpolants," *tests and proofs*, pp. 151–168, 2016.
[35] M. Pradel, R. Qian, E. Meijer, and S. Chandra, "Scaffle : Bug Localization on Millions of Files," 2020.
[36] S. Roychowdhury and S. Khurshid, "Software fault localization using feature selection," *Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering*, pp. 11–18, 2011.

[37] R. van Tonder, J. Kotheimer, and C. le Goues, "Semantic crash bucketing," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 612–622.

[38] S. Wang, F. Khomh, and Y. Zou, "Improving bug localization using correlations in crash reports," in *2013 10th Working Conference on Mining Software Repositories (MSR)*, 2013, pp. 247–256.

[39] M. Weiser, "Program slicing," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 352–357, 1984.

[40] W. E. Wong and Y. Qi, "An execution slice and inter-block data dependency-based approach for fault localization," in *Asia-pacific Software Engineering Conference*, 2004.

[41] ——, *Effective program debugging based on execution slices and inter-block data dependency*. Elsevier Science Inc., 2006.

[42] X. Y. Xie, F. C. Kuo, T. Y. Chen, S. Yoo, and M. Harman, "Provably optimal and human-competitive results in sbse for spectrum based fault localisation," in *International Symposium on Search Based Software Engineering*, 2013.

[43] J. Xu, D. Mu, X. Xing, P. Liu, P. Chen, and B. Mao, "PoMP: Postmortem program analysis with hardware-enhanced post-crash artifacts," *Proceedings of the 26th USENIX Security Symposium*, pp. 17–32, 2017.

[44] J. Xu, R. Chen, and Z. Du, "Probabilistic reasoning in diagnosing causes of program failures," *Software Testing Verification and Reliability*, vol. 26, no. 3, 2016.

[45] K. Xu, J. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhudinov, R. Zemel, and Y. Bengio, "Show, attend and tell: Neural image caption generation with visual attention," in *International conference on machine learning*, 2015, pp. 2048–2057.

[46] X. Ye, R. Bunescu, and C. Liu, "Learning to rank relevant files for bug reports using domain knowledge," in *Acm Sigsoft International Symposium on Foundations of Software Engineering*, 2014.

[47] S. Yoo, "Evolving human competitive spectra-based fault localisation techniques," in *SSBSE'12 Proceedings of the 4th international conference on Search Based Software Engineering*, 2012, pp. 244–258.

[48] F. R. Zakani, K. Arhid, M. Bouksim, T. Gadi, and M. Aboulfatah, "Kulczynski similarity index for objective evaluation of mesh segmentation algorithms," in *2016 5th International Conference on Multimedia Computing and Systems (ICMCS)*, 2016.

[49] M. Zalewski, "American fuzzy lop," *URL: http://lcamtuf. coredump. cx/afl*, 2017.

[50] S. Zhang and C. Zhang, "Software bug localization with markov logic," *Companion Proceedings of the 36th International Conference on Software Engineering*, pp. 424–427, 2014.

[51] X. Zhang, N. Gupta, and R. Gupta, "Pruning dynamic slices with confidence," *Acm Sigplan Notices*, vol. 41, no. 6, pp. 169–180, 2006.

[52] ——, "A study of effectiveness of dynamic slicing in locating real faults," *Empirical Software Engineering*, vol. 12, no. 2, pp. 143–160, 2007.

[53] X. Zhang, S. Tallam, N. Gupta, and R. Gupta, "Towards locating execution omission errors," vol. 42, 06 2007, pp. 415–424.