Separating the Wheat from the Chaff: Using Indexing and Sub-Sequence Mining Techniques to Identify Related Crashes During Bug Triage

Kedrian James[1], Yufei Du[1], Sanjeev Das[2], and Fabian Monrose[3]
[1]UNC Chapel Hill, Chapel Hill, NC, USA
[2]Independent Researcher, CA, USA
[3]Georgia Institute of Technology, Atlanta, GA, USA

***Abstract*—Bug triaging entails a laborious process wherein triagers spend time examining new bug reports, localizing the bugs, and assigning them to the appropriate developer(s) to fix the bugs. In recent years, the adoption of automated software testing techniques (*e.g.,* fuzzing) further complicates the process because bug hunters can submit an overwhelming number of reports in a short period. To lessen these pain points, we present an approach that extracts a fingerprint from crash information within a bug report, and returns a group of bugs with similar behaviors. Our approach uses symptoms of the crash to create a robust fingerprint, and leverages MinHashing and Locality Sensitive Hashing to match crashes, as well as a sequential pattern mining algorithm to find frequent closed sequences among bugs. Our evaluation shows that our approach outperforms contemporary approaches (*e.g.,* finding previously unknown duplicates among 81 CVEs), and saves triagers time and effort.**

Keywords*—*Security; bug fixing

I. Introduction

Over the past decade, automated software testing techniques (*e.g.,* fuzz testing) have risen in popularity due to their ease of use and effectiveness in finding bugs. To date, such automated software testing techniques have successfully discovered hundreds of exploitable bugs in widely used software. As a result, project maintainers often receive bug reports on a daily basis, with popular large-scale projects, like *Gnome*, receiving over 120 submissions daily [41].

Inevitably, many of the reported bugs are duplicates of existing bugs due to the fact that bug hunters used widely different tools and practices, and often end up submitting reports for the same flaws [41, 7, 21]. While contemporary deduplication approaches (*e.g.,* stack hashing, edge coverage) can in most cases identify bugs that are exact duplicates of existing bug, they are not effective in case of inexact duplicates [29, 33] — bugs that share similar characteristics and causes with a known bug but are not completely identical to the known bug. The deluge of reported bugs is a problem because the task of identifying duplicate bugs, and categorizing the relationship among bugs (*e.g.,* different bugs in the same function), is generally left to a triager. The process is time consuming and tedious if for no other reason than it demands manually searching through a large datastore to infer relationships. Anvik et al. [6] noted that in 2005, even before fuzzing became as popular as it is today, the *Eclipse* project required two man-hours daily just to address duplicates. For large projects like *Mozilla* — where duplicates account for 30% of the total bugs reported [6] — the time wasted quickly adds up.

Obviously, identifying and analyzing the relationship among reported bugs is important for several reasons. First, identifying duplicate bugs can significantly improve the productivity of triagers. Jeong et al. [27] shed light on how time-consuming the triaging process can be, and notes that from their analysis of 45,000 bug reports from *Eclipse*, they found that on average, a bug takes 16.7 days to receive the first action and 23.6 days to be assigned. The assignment takes time as triagers must check each report, understand the cause of the bug, assign a priority to the bug, and then match each bug to the proper developer. With duplicate bugs identified and grouped, triagers only need to inspect a subset of the reports. Second, analyzing the relationship among bugs can also significantly improve downstream tasks like root cause analysis [24]. Although duplicate bugs are usually unwanted, sometimes they could assist the analysis. Bettenburg et al. [8] shows that identifying duplicates can sometimes provide more perspective and pointers about the source of a bug. For instance, if a newly reported bug is an inexact duplicate of a previously fixed bug, then it means the patch was incomplete, and the bug could still be exploited in other ways.

To provide a pragmatic solution that addresses the dearth of tools in this area, we present *CrashSearch*. CrashSearch uses the symptoms of a crash available in a bug report to create a fingerprint that it uses to search for bugs with similar behaviors, and leverages sequential pattern mining to effectively showcase relationships among duplicate bugs.

Specifically, our contributions include:

- A novel hybrid approach using MinHashing [11] (to estimate the similarity between bugs based on features extracted from their bug reports) and Locality Sensitive Hashing [23] (to store groups of similar bugs in the same locality in a datastore).
- The application of the BI-Directional Extension (BIDE) algorithm [48] on stack backtraces to categorize the relationship (at both function and line granularities) among groups of similar bugs.
- A case study of how *CrashSearch* was used to triage

crashes generated during a fuzzing campaign.
- An analysis of discovered similarities among 81 Common Vulnerabilities and Exposures (CVEs) submitted over the past several years.

II. Background

a) Bug Triaging.

Large scale software projects such as *PHP* and *Mozilla* use bug tracking systems such as Bugzilla [1] to facilitate the submission and management of bug reports. When reporting a bug, the submitter is required to provide a summary of the bug, steps to reproduce the bug, and search the bug datastore for duplicate bugs [1]. Due to the inefficiencies of contemporary deduplication tools at finding unique bugs [29, 33] and diversity in bug report construction [8], duplicate bugs are often reported. Subsequent to bug submission, a triager generally conducts a manual bug triaging process. Triaging is the process of analyzing the reported bugs. Triaging typically involves three steps: deduplication, prioritization, and assignment. During deduplication, a triager generally conducts a manual comparison between a newly reported bug and existing bugs to identify duplicates [5, 8]. Whenever duplicate bugs are detected, the triager marks them as duplicates and adds a reference to the original bug report [5, 8]. Following deduplication, bugs are assigned to a responsible developer along with a priority [5, 8]. Once assigned, bug fixes are typically prioritized based on the potential damage the flaw could cause [44].

b) Similarity Comparison.

The problem of determining the similarity among large datasets is by no means new. Previous works [11, 23, 16] have studied techniques to find exact and inexact duplicate items. These techniques have been applied to various fields within computer security, including (but not limited to) malware analysis [39] and reused function detection in program binaries [4]. Within these domains, the Jaccard similarity measure is one of the most widely used metrics to compute the similarity between objects.

The Jaccard similarity is simply the ratio of the size of objects' intersection to the size of their union [32]. While effective, computing the Jaccard similarity is expensive, and it requires pairwise comparisons. Therefore, it is not suitable in scenarios where there are a large number of objects to be compared. Instead, the MinHash algorithm [11] is used to efficiently approximate the Jaccard similarity between two objects by hashing the elements of each object using N hash functions and retaining only the minimum hash values for each hash function. The estimated Jaccard similarity is the ratio of the intersection of hash values for both objects, divided by the total hash functions N. MinHash improves efficiency because it does not require pairwise comparison among all the elements of an object and each object is represented as a fixed size integer array of size N. Moreover, MinHash supports Locality Sensitive Hashing (LSH) which is a technique that improves the scalability of nearest neighbor search for large datasets by hashing similar objects to the same locality.

III. Related Work

Brodie et al. [13] were the first to discuss the need for a practical system that identifies known software problems. To that end, the authors applied a custom incremental machine learning approach that generates a signature to find a "best" matched sequence of consecutive function names among stack backtraces. Additionally, they adopted the Needleman-Wunsch algorithm for matching stack backtraces and then applied `top-`k indexing to eliminate pairwise comparisons. Modani et al. [35] extended Brodie et al. [12] to include three string matching algorithms for stack backtrace similarity, namely Edit Distance, Longest Common Subsequence and Prefix Matching, and found that the `top-`k indexing scheme outperformed existing [21, 7, 35] solutions.

More distantly related are approaches that leverage natural language processing techniques [30, 51, 26, 38] to detect duplicate bug reports. These techniques, however, are focused on clustering of reports based on textual data in the report, and serve as a quick way to group reports. For bugs generated from crashes, a textual description is not always available to compare similarities. More importantly, they *do not help in pinpointing the likely expressions* (both sinks and sources) responsible for the crash. An exception is the work of Wang et al. [50], that combines stack backtrace and natural language features to detect duplicate reports. Their approach uses the vector space model, and each element in the vector is calculated using Term Frequency-Inverse Document Frequency (TF-IDF). The stack backtrace and the natural language features are stored in separate vectors, and they are compared separately during classification. Unfortunately, since most toolchains (e.g., fuzzers) do *not provide bug descriptions of crashes written in natural language*, approaches that rely on such data are fairly limited in practice.

Bartz et al. [7] later proposed a machine learning-based approach to identify similar failures. The authors used three categorical features, namely exception code, process name, and event type along with similarity measures of the stack backtrace computed using edit distance to fit a logistic probability model. To eliminate pairwise comparison, the authors adopted the `top-`k indexing scheme to provide an initial list of candidate failures. Likewise, Dang et al. [19] proposed an approach dubbed `ReBucket` for clustering duplicate crash reports based on stack backtrace similarity. The `ReBucket` approach uses a Position Dependent Model that computes the similarity of two stack backtraces based on the number of functions on both stack backtraces, the distance of the functions from the top stack frame, and offset distance between the match functions. Stack backtraces are

Approach	Features	Similarity Algorithm	Large Scale Storage	Group Analysis
Bartz et al. [7]	Call Stack, Exception Code, Process Name, Event Type	Edit Distance Machine Learning (logistic probability)	Top-k Indexing	None
Brodie et al. [13]	Call Stack	Best Matched Sequence ML Model	None	None
Brodie et al. [12]	Call Stack	Needleman-Wunsch algorithm	Top-k Indexing	None
Castelluccio et al. [15]	Platform, Version, Addons, Modules, CPU info	None	None	Contrast-set Mining
Dang et al. [19]	Call Stack	Position Dependent Model (PDM)	None	None
Dhaliwal et al. [21]	Call Stack	Edit distance	None	None
Khvorov et al. [28]	Call Stack	Machine Learning (long short-term memory)	None	None
Lerch and Mezini [31]	Call Stack	Term Frequency-Inverse Document Frequency	None	None
Modani et al. [35]	Call Stack	Levenshtein Distance Longest Common Subsequence Prefix Matching	Top-k Indexing Inverted Indexing	None
Sabor et al. [41]	Call Stack, Component, Severity	Cosine Similarity	None	None
Vasiliev et al. [46]	Call Stack	Term Frequency-Inverse Document Frequency Edit Distance	None	None
Wang et al. [50]	Call Stack, Bug Summary Text Bug Description Text	Vector Space Model Term Frequency-Inverse Document Frequency	None	None
CrashSearch (Our Approach)	Signal, Bug Type, Call Stack, Crash Function, Crash Line	MinHashing	Locality Sensitive Hashing (LSH)	BI-Directional Extension (BIDE)

TABLE I: Comparison of contemporary crash similarity approaches

clustered based on their similarity measures.

Lerch and Mezini [31] used TF-IDF for finding duplicate bug reports for the Eclipse project. The similarity of two stack backtraces is determined by the number of similar functions they have. In a similar fashion, Sabor et al. [41] introduced DURFEX, which is an approach for finding duplicate bug reports for Java crashes. The authors combined the stack backtrace similarity along with the similarity of two non-textual fields. Each trace is represented as a feature vector and similarity is measured using cosine similarity. In analyzing bug similarity, Castelluccio et al. [15] apply a contrast-set mining technique to find significant deviations among groups of crashes. Closeness is computed relative to a user-specified threshold. Different from this approach, we perform sequential pattern mining on stack backtraces to analyze the relationship among duplicate crashes. To improve pairwise comparisons among stack traces, Rodrigues et al. [40] proposed an approach that uses a sequence alignment algorithm to compute the similarity score between stack traces in linear time. By contrast, *we forego the need to conduct pairwise comparisons* among all stack traces via applications of novel indexing.

Several works [17, 52, 18] attempt to infer the root cause of bugs by augmenting the crash deduplication process. However, these root cause analysis techniques *impose more operational requirements on the triager than we consider practical* (*e.g.,* having the binary, a core dump, and a capability to collect instruction traces) in most settings. Moreover, performing root cause analysis on every crash requires human verification – a task which could be made less burdensome by only performing more laborious root cause analyses on the unique reports filtered using the techniques described herein.

IV. Approach

We designed *CrashSearch* for the specific purpose of efficiently mapping a reported bug to related known bugs within a bug datastore, including both exact and inexact duplicates. We also provide a way to examine relationships among similar bugs. The motivation stems from the fact that despite many years of research in the academic community, crash triaging still remains an important problem in dire need of practical solutions. Motivating applications abound. For instance, in a recent study of 0-day exploits, Stone [43] found that 25% of 0-day exploits found in 2020 were variants of previously found vulnerabilities, and within these vulnerabilities, half of the bugs were fixed incompletely when they were first discovered. Recently, Vyukov [47] emphasized that the Linux kernel team desperately needs more developers to help with triaging of bugs.

In response to that need, our solution encompasses the four components shown in Figure 1. The report processing component ① extracts the features from a bug report and generates data structures required for later components, including the fingerprint of the bug and the LSH indexes. The insertion component ② uses the indexes to efficiently store fingerprints of bugs. The query component ③ takes the indexes to retrieve similar candidates from the datastore and performs similarity comparisons. The final component ④ performs pattern mining to discover the relationship among bugs.

A. Component ① Report Processing

a) Feature Extraction.

The features CrashSearch uses are given in Table II. We treat the signal, bug type, crash function, and crashing line as strings. For the backtrace, we group consecutive function names as *bi-gram* and select each pair as a feature. It is

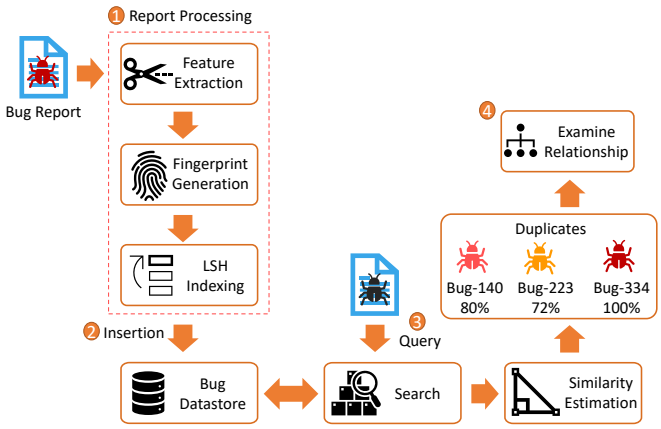

Figure 1: Overall workflow of CrashSearch

typical for bug reports describing crashing bugs to contain the required information, and other textual descriptions which we do not use. CrashSearch uses a script to extract the features from the output of the crash in the bug report. In a situation where a bug report is missing some of the required information, the script is also capable of generating the information by running the program with the GDB debugger and AddressSanitizer [42], if the bug report includes a reproducible crashing input.

Full Data	Encoding	Feature Example
Signal	String	SIGSEGV, SIGABRT
Bug Type	String	Use-after-free, heap-overflow
Crash Function	String	foo()
Backtrace	Bi-gram	(main,foo),(foo,bar),(bar,baz)
Crashing Line	String	value = *ptr++;

TABLE II: Features used for generating bug fingerprint

b) Fingerprint Generation.

Once features are collected, we use the MinHash [11] algorithm to generate a fingerprint for each bug. We apply N different hash functions to generate the fingerprint for a bug, where each hash function generates a signature that is part of the fingerprint. In our implementation, we empirically selected `N=256` hash functions to have a fair balance of performance and error rate when estimating Jaccard similarity. The estimation is based on the formula $(1/\sqrt{N})$. This means that our estimate in Jaccard similarity will be off by 0.06%. We use a non-cryptographic hash function, `MurmurHash`, for efficient hashed-based lookups.

c) Index Generation.

In order to reduce the amount of comparisons, CrashSearch utilizes LSH during insertion and query such that it only needs to compare a fingerprint to a small subset of fingerprints in the datastore. CrashSearch computes the LSH indexes before insertion or query. CrashSearch divides the signatures of each fingerprint into b bands, where each band contains r rows of signatures. For each band, CrashSearch generates the index by hashing it using the LSH hash function, which allows similar values to collide. The indexes are later used in the insertion component and the query component. §V-A1 describes how b and r are chosen in practice.

B. Component ② Insertion

We use LSH to implement efficient insertion of bugs into our datastore. Specifically, LSH groups items into different buckets, with items in the same bucket sharing similar values [23]. For each band of a fingerprint, CrashSearch inserts it into a bucket in the datastore. This means that bands in the same bucket share similar signatures, and new fingerprints can be inserted without re-hashing all existing fingerprints.

C. Component ③ Query

When querying for potentially similar bugs, we use the LSH indexes to determine the bucket for each band of the report's fingerprint. CrashSearch considers all fingerprints whose bands collide with a band of the query fingerprint to be candidates. For each candidate, we compare the signatures of the candidate to the signatures of the query fingerprint in the collided band. Specifically, we estimate Jaccard similarity as $JS(Q,C) = \frac{|Q_M \cap C_M|}{N}$, where Q_M represents the MinHashes in the query fingerprint, C_M represents the MinHashes in the candidate fingerprint and N is the number of hash functions. After comparing with all the candidate fingerprints, we return a group (if any) of candidates along with the corresponding bug description and Jaccard similarity score.

D. Component ④ Examine Relationships

In addition to finding exact and inexact duplicate bug reports, CrashSearch can be used to examine the relationships among duplicate reports. For that, we leverage the BI-Directional Extension (BIDE) algorithm [1], which is a type of sequential pattern mining algorithm used for mining *frequent closed ordered subsets* [48]. The frequency is based on a user-defined threshold called the min support, which is the minimum number of sequences in which the given pattern occurs. Besides computing common function call sequences from the stack backtrace, we also extract shared sequences at the line-level granularity. We have found this capability to be extremely helpful as it allows a triager to gain better insights into the relationships among bugs over time.

V. Real-world Evaluation

To evaluate the effectiveness of our approach, we designed experiments that centered on gaining insights relative to four questions:

RQ1 *How well does CrashSearch identify duplicates of known bugs, compared to existing solutions?*

[1]https://github.com/RonaldYu/bide-algorithm

RQ2 *Is CrashSearch efficient enough to support large software projects?*

RQ3 *Can CrashSearch help triagers more easily find relationships among bugs?*

RQ4 *Can CrashSearch effectively reduce the number of crashes triagers have to analyze?*

a) Assumptions and Experimental Setup.

We assume the triager(s) maintains a datastore of crash fingerprints. As CrashSearch supports inserting new fingerprints to a datastore, the maintainer can utilize that capability to initialize a new datastore with information from existing bug reports. Experiments were conducted on a system with an Intel(R) Core(TM) i7-4790 CPU processor and 16 GB of memory. For the first set of experiments, we established ground truth for inexact duplicate reports by generating crashes for each bug using the AFL fuzzer [53] configured in crash exploration mode. The crash exploration mode takes a crashing test case and mutates it to generate new crashes that trigger the same bug but traverse new code paths. The same methodology has been applied in prior studies on crash triaging [45] and bug localization [9]. Each fuzzing run was conducted for a period of 24 hours. For the experiments in Section V-D, we use different fuzzing tools to mimic the vulnerability discovery process and practices employed by different groups prior to submitting their reports to a project maintainer. These experiments ran in a virtualized environment on an Intel(R) Xeon CPU E5-2630 v4 processor and 128 GB memory. Experiments with different fuzzers were run on separate days.

A. RQ1: How well does CrashSearch identify duplicates of known bugs, compared to existing solutions?

To assess the accuracy of our approach, we compare CrashSearch with techniques that have been widely adopted for crash triaging by security researchers and practitioners, and state-of-the-art approaches in indexing and in similarity search. Specifically, we compare the LSH indexing component of CrashSearch with the top-k indexing scheme as it is one of the most widely used techniques [12, 35, 14] for matching fingerprints. Second, we compare the MinHash-based component of CrashSearch with the leading contenders (*i.e.,* prefix match [35], TraceSim [46], and major and minor stack hashing [29, 33, 22]) commonly used for deduplication.

		Duplicates		
Program	**Version Count**	**Exact Test Set**	**Inexact Test Set**	**Negative Test Set**
JasPer	2	5	160	165
libarchive	2	4	367	371
libjpeg-turbo	3	4	113	117
libtiff	3	5	245	250
libxml2	3	5	129	134
pcre	4	4	315	319
Total		27	1329	1356

TABLE III: Dataset used for the experiments targeting **RQ1**

Table III lists the programs and the statistics of the fingerprints we use for this experiment. We choose these programs because they are popular open-source programs, and more importantly, prior works [45, 20, 52] have documented bugs in them along with proof of concept inputs. We use these documented bugs to generate the fingerprints for this experiment. For each bug, we generate inexact duplicate bugs using the AFL fuzzer [53] configured in crash exploration mode. Then, we insert the documented bugs to the datastore to build our reference fingerprint set. For each program, we exclude one documented bug when building the reference fingerprint set. This excluded bug, and all the inexact duplicates derived from this bug, form our negative test set (*i.e.,* fingerprints that do not have a related fingerprint in the reference set). All other documented bugs and their inexact duplicates form our positive test set. As the documented bugs are also in the reference set, these bugs form the exact test set while the inexact duplicates of these bugs form the inexact test set.

Given that our positive test set is significantly larger than our negative test set, we take an additional step of balancing our test sets. For each program, we randomly remove the bugs in the inexact test set such that we have the same number of bugs in the positive test set (including both the exact and the inexact test sets) and the negative test set. To minimize bias, we select crashes with equal probability, and ensure that crashes for each bug are as close to the mean of the total crashes in the positive test set. Following this procedure, each experiment is run 20 times, and we report the average scores. In total, our collection includes 27 bugs in the exact test set, 1,329 bugs in the inexact test set, and 1,356 bugs in the negative test set.

1) Effectiveness of Indexing:

We first examine the effectiveness of our indexing approach. For this experiment, our goal is to determine whether our indexing technique can index similar fingerprints to the same bucket. For comparison, we compare our LSH approach with top-k, a common indexing technique used by many previous works [7, 12, 35].

For LSH, values for b and r must be chosen appropriately. In tuning these parameters, we ensure that $b \times r = 256$ (*i.e.,* the number of MinHash signatures for each fingerprint). Additionally, we select values for b and r that allow us to correctly retrieve candidate fingerprints. We compute the estimated similarity between a pair of fingerprints and the probability that LSH will consider a pair of fingerprints to be truly similar as $1-(1-s^r)^b$, where s denotes the estimated Jaccard similarity.

Intuitively, the optimal values for b and r should find a balance between the Jaccard similarity and the probability that LSH deems a pair of fingerprints similar. If the sensitivity is too low (*i.e.,* LSH only finds a pair of fingerprints to be similar when their Jaccard similarity is extremely high), then we could easily miss inexact duplicate fingerprints. On the other hand, if the sensitivity is too high (*i.e.,* LSH finds a

pair of fingerprints to be similar even when their Jaccard similarity is extremely low), then we need to perform a large amount of unnecessary comparisons of fingerprints that do not match at all. Extensive evaluations from parameter tuning show that $b = 64, r = 4$ strikes the best balance.

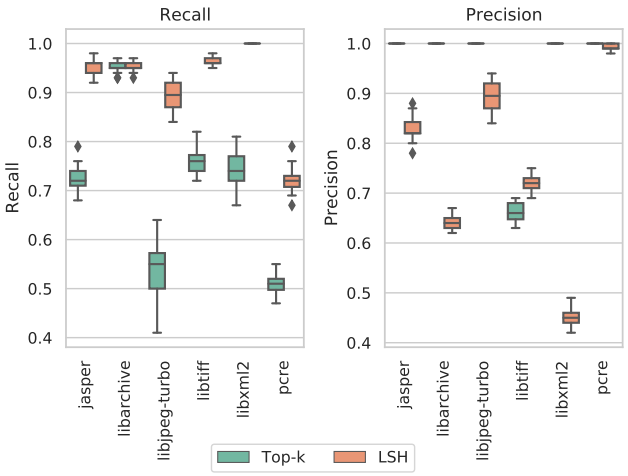

Figure 2: Performance of Top-k vs LSH indexing

Figure 2 shows the results of the top-k indexing and LSH 64 averaged over 20 runs. In every case except `libarchive`, we attained higher recall than top-k indexing. While the top-k algorithm performs better on certain benchmarks in terms of precision, for us, recall is more important at this stage because candidates in the same bucket are then passed to the similarity comparison phase where false positives are filtered.

2) Effectiveness in Filtering Duplicates:

While our indexing component, LSH, can bucket similar fingerprints into the same bucket with high recall, it also includes some unrelated fingerprints to the same bucket, as evidenced by the lower precision. Therefore, our MinHashing component then takes the fingerprints in the same bucket, determines the similarity between fingerprints, and reports fingerprints that are similar while filtering out unrelated fingerprints.

In this experiment, we compare our MinHashing approach with previous similarity comparison approaches, including top-k prefix match [35, 12], TraceSim [46], major and minor stack hashing [29, 33, 22]. Although TraceSim is open sourced[2], the implementation of the machine learning based hyperparameter generation is neither publicly available nor documented. Thus, to allow for a fair comparison, for each program, we run TraceSim with all possible combinations of the hyperparameters α, β, and γ, as well as the threshold value, in the range of `0.1` and `1`, with a step of `0.1`. We select the hyperparameters and threshold values that yield the highest F1 score for each program.

[2]https://github.com/traceSimSubmission/trace-sim

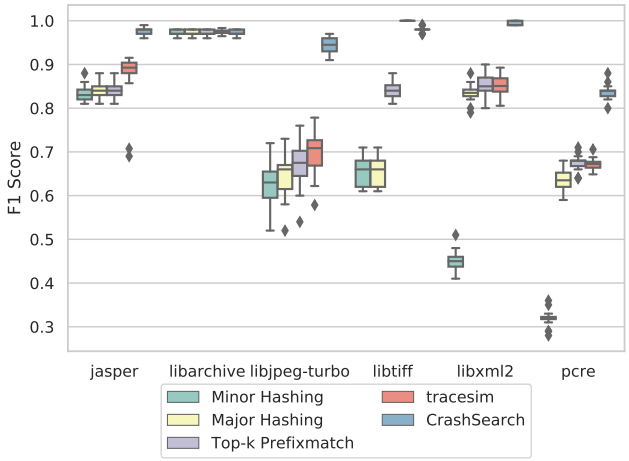

Figure 3: Comparison of contemporary crash similarity approaches

Figure 3 shows the results averaged over 20 runs. For the similarity among exact duplicates, all approaches perform equally well having an F1 score of 1. However, in the more realistic experiments that use inexact duplicates during the query phase, CrashSearch outperforms the others in four of the six cases: `JasPer`, `libjpeg-turbo`, `libxml2` and `pcre`. For the other two, CrashSearch and TraceSim perform equally well, outperforming the others on `libtiff`. Among these programs, CrashSearch achieves significantly better results than other approaches in `libjpeg-turbo` and `libxml2`. These two programs contain bug instances that crash at different functions (*e.g.,CVE-2018-11213* for `libjpeg-turbo` and *CVE-2017-9049* for `libxml2`), so approaches that only use stack backtraces struggle to match the inexact duplicate fingerprints of these bugs. The results also show that our MinHashing component can detect and filter out all false positives that the LSH component generates, as our precision increases compared to the precision of LSH indexing, for all programs. A closer inspection of the `libxml2` bug relationships (not shown) reveals that while four CVEs have significantly different stack backtraces, that is due to a macro used by four separate functions. Therefore, all functions that utilize the macro are affected, and so each seemingly appears to be a different bug. TraceSim fails to identify 2 of the CVEs as similar. Since we utilize additional features besides the stack backtrace (*i.e.,* the signal, the bug type, and the crashing line of code), we are able to succeed where TraceSim fails. Additionally, TraceSim induces 3 false positives.

3) **Insights and lessons learned:**

Overall, the experiments show that the combined usage of LSH indexing and MinHash worked exceedingly well for crash triaging. For inexact duplicates, on average, the F1 score of CrashSearch is 11% more than TraceSim, 15% more than top-k prefix match, 19% more than major hashing, and

31% more than minor hashing. There is, however, room for improvement; for example, with programs like `pcre` that heavily use macros.

B. RQ2: Is CrashSearch efficient enough to support large software projects?

To explore our performance in more operational settings that could, for example, include tens of thousands of known bugs [47], we conduct experiments that artificially inflate the size of the database to compare the scalability with state-of-the-art approaches like TraceSim [46] and top-*k* prefix match [35].

To select the query fingerprints, we use a subset of the test fingerprints used in §V-A2. Specifically, for each of the six real world programs, we select the fingerprint whose stack backtrace contains the smallest amount of stack frames, the fingerprint whose stack backtrace contains the largest amount of stack frames, the fingerprint whose stack backtrace contains the smallest amount of characters, and the fingerprint whose stack backtrace contains the largest amount of characters. From the set, we remove duplicates as some fingerprints could both have the smallest amount of stack frames and the smallest amount of characters, and finalize a set of 30 fingerprints in total. We then synthetically duplicate this set of fingerprints into different configurations by artificially altering features such as the bug type and the crashing function, generating databases with 300 fingerprints, 3,000 fingerprints, 10,020 fingerprints, and 30,000 fingerprints. For each configuration, we run the approach five times and report the average.

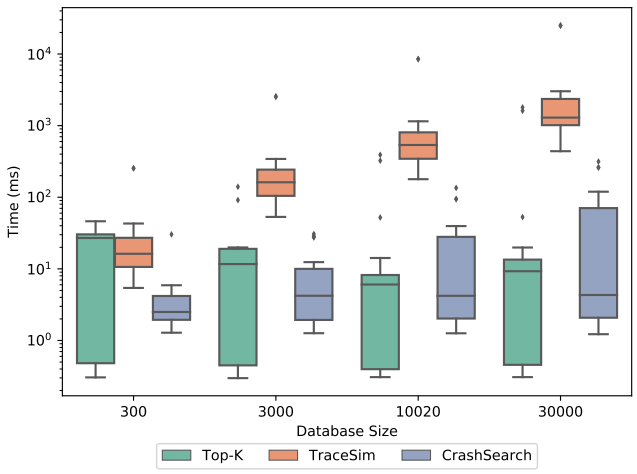

Figure 4: Query time versus database cardinality

The results are shown in Figure 4. For databases with up to 300 reference fingerprints, the overhead between TraceSim and CrashSearch is negligible. However, as more reference fingerprints are inserted into the database, the performance of TraceSim degrades significantly. The results of TraceSim also contains an outlier that takes significantly longer time (25.19 seconds in a database with 30,000 fingerprints) than others. The fingerprint causing this outlier contains a large stack backtrace (with 251 stack frames) due to recursion.

1) **Insights and lessons learned:**

The performance benefits of CrashSearch are not surprising given the well-known efficiency of the underlying techniques we choose to implement. Additionally, though it may seem that the query time for TraceSim is acceptable, this is not necessarily the case because, even after deduplication, a fuzzing campaign may produce hundreds of crashing inputs in a short period of time [29]. Moreover, in settings where one might choose to run triaging at the time of fuzzing (*e.g.,* as suggested by Das et al. [20] or, more generally, to assist with directed fuzzing [10]), then the increased overhead would lead to unacceptable slowdowns. Lastly, while top-*k* prefix match runs fast, it does so at the cost of lower accuracy, and falls short of the ultimate goal: reducing the amount of manual effort expended by analysts. Our experiment shows that even for large projects, the query time for CrashSearch would still be acceptable.

			Average	
Program	**# Bugs**	**Duplicate Groups**	**Bugs**	**Shared Sequences**
JasPer	32	28	4	5 ⊂ 9
libarchive	21	14	2	6 ⊂ 9
libxml2	38	15	3	7 ⊂ 15
FFmpeg	49	38	6	11 ⊂ 19
ImageMagick	106	100	7	8 ⊂ 13
PHP	116	18	6	8 ⊂ 17
Wireshark	114	79	8	21 ⊂ 38

TABLE IV: Duplicate bug groups and shared sequences

C. RQ3: Can CrashSearch help triagers more easily find relationships among bugs?

We now highlight CrashSearch's ability to help triagers analyze the relationships among bugs. For this experiment, we take a more in-depth look at `JasPer`, `libarchive`, and `libxml2` (from the dataset in §V-A) as well as four other popular open-source programs: `ImageMagick`, `PHP`, `FFmpeg`, and `Wireshark`. We choose to study these eight programs because we can amass bug reports of these programs spanning multiple years. Note that, since previous work focuses solely on deduplication and does not include similar techniques for understanding the relationships among bugs (as shown in Table I), we do not have a state-of-the-art to compare in this experiment.

To analyze the relationships, we first use the query component of CrashSearch with a similarity threshold of $\frac{1}{3}$ to extract groups of likely duplicate bugs solely based on the features extracted from their bug reports. Next, we use the relationship component of our approach, which leverages BIDE, to extract shared function call sequences from the stack backtraces for each group. We selected a conservative threshold of $\frac{1}{3}$ based on experiments in §V-A to ensure that

we do not miss potential duplicates. Moreover, in real-world settings, a triager may not have ground truth readily available in order to find the optimal threshold. Although using a conservative configuration can lead to false positives, these will be eliminated later by the analysis component.

Table IV shows the number of bugs in our dataset, the number of groups of duplicate bugs we find, the average amount of bugs in each group, the average shared call sequences for all bugs in the group. In general, we find that many bugs in the same group have relationships in their call sequences. For instance, among all groups for `PHP`, bugs share on average 8 of 17 function call sequences. Similarly, for `ImageMagick` on average bugs share 8 of 13 function call sequences. Moreover, the same pattern is evident for `JasPer`. As an example, Table V list the reduced groups of duplicate bugs we find for `JasPer` and `ImageMagick` after using BIDE to examine the relationships. To shed further light on our findings and lessons, we focus on `JasPer` and `ImageMagick` as these two programs each contain bugs with strong relationships.

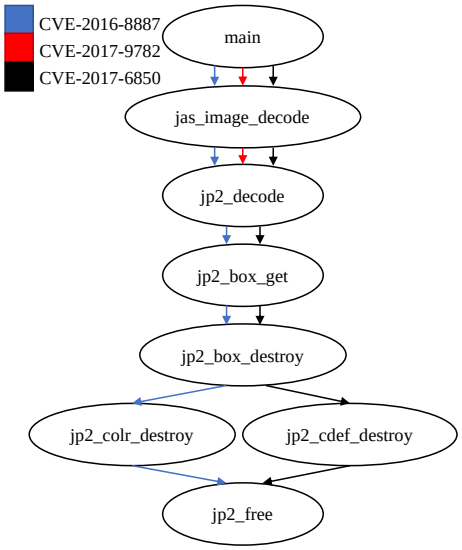

Figure 5: Example of relationship among bugs in JasPer

a) Case Study I:

We analyzed 8 bugs for Jasper (*CVE-2017-9782*, *CVE-2018-19542*, *CVE-2018-19543*, *CVE-2021-26926*, *CVE-2021-26927*, *CVE-2021-3272*, *CVE-2021-3443*, and *CVE-2021-3467*) submitted over a 5 year period that our query component correctly flagged as inexact duplicates of each other. Using the analysis component we found that these bugs all share the same crashing sequence (see *CVE-2017-9782* in Figure 5). Interestingly, 5 of these bugs were repeatedly submitted by different testers over a few months. It was not until the last submission that a contributor discovered that all 5 bugs were related and could be narrowed down to one root cause [2]. Even then, the contributor did not realize the connections to 3 other bugs namely, *CVE-2017-9782*, *CVE-2018-19542*, and *CVE-2018-19543*.

We also found strong connections between *CVE-2016-8887* and *CVE-2017-6850*, submitted in 2016 and 2017 respectively. Specifically, these bugs share 5 of 6 function sequences. Figure 5 illustrates the call sequences of these bugs. To confirm that these bugs were indeed duplicates, we manually analyzed the bugs using both static and dynamic analysis. For that painstaking process, we used contemporary tools (including `Mozilla rr` [36], `GDB debugger`, and `Zelos CrasHD` [54]), to perform debugging and dataflow analysis. We discovered that both bugs do share the same underlying issue: the use of an uninitialized variable.

			Verification	
Program	Bugs	Duplicate Type	Static Analysis	Dynamic Analysis
ImageMagick	CVE-2016-7515 CVE-2016-7519	Exact	●	●
	CVE-2016-7523 CVE-2016-7524	Exact	●	●
	CVE-2016-7518 CVE-2017-6500	Exact	●	●
	CVE-2016-8862 CVE-2016-8866 CVE-2017-7275	Exact	●	○
	CVE-2019-10650 CVE-2019-11597 CVE-2019-15141	Exact	●	○
	CVE-2019-13295 CVE-2019-13297	Exact	●	●
	CVE-2019-13302 CVE-2019-13308	Exact	●	●
	CVE-2019-13304 CVE-2019-13305 CVE-2019-13306	Exact	●	●
	CVE-2018-16412 CVE-2018-16413	Inexact	●	●
	CVE-2016-7514 CVE-2016-7525	Inexact	●	●
	CVE-2018-11625 CVE-2019-11598	Inexact	●	●
	CVE-2017-11533 CVE-2019-19948	Inexact	●	●
	CVE-2016-7515 CVE-2016-7518 CVE-2017-6500	Inexact	●	●
JasPer	CVE-2017-9782 CVE-2018-19542 CVE-2018-19543 CVE-2021-3272 CVE-2021-3443 CVE-2021-3467 CVE-2021-26926 CVE-2021-26927	Inexact	●	○
	CVE-2016-8887 CVE-2017-6850	Inexact	●	●
	CVE-2016-10251 CVE-2017-6852	Inexact	●	○
	CVE-2017-5499 CVE-2017-5500	Inexact	●	○

The ● symbol implies that the specific analysis was performed, while the ○ symbol means otherwise. Static analysis is done via source code inspection, while dynamic analysis is a combination of debugging and dataflow analysis.

TABLE V: Duplicates flagged for `JasPer` and `ImageMagick`.

b) Case Study II:

For ImageMagick, we examine the relationship among 4 bugs reported by 2 different testers in 2016 and 2019. Figure 6 shows the graph constructed from the stack backtraces.

Using *CVE-2016-7523* as a query, we retrieved *CVE-2016-7524*, *CVE-2019-13295*, and *CVE-2019-13297* with 100%, 35% and 35% similarity scores respectively. Leveraging the analysis component, we discovered that in addition to *CVE-2016-7524* and *CVE-2016-7524*, *CVE-2016-7524* and *CVE-2019-13295* also shared exact sequences. As a result, we could easily isolate and analyze the two sets of bugs separately to confirm that they were duplicates. We found that *CVE-2016-7523* and *CVE-2016-7524* were submitted by the same tester with two test cases that crash the program at different line numbers. Digging deeper we found that the crashing line of code for both bugs is the same because the developer replicated the code in different locations of the source file. Similarly, we confirmed that *CVE-2019-13295*, and *CVE-2019-13297* shared the same root cause and were duplicates of *CVE-2016-7523*.

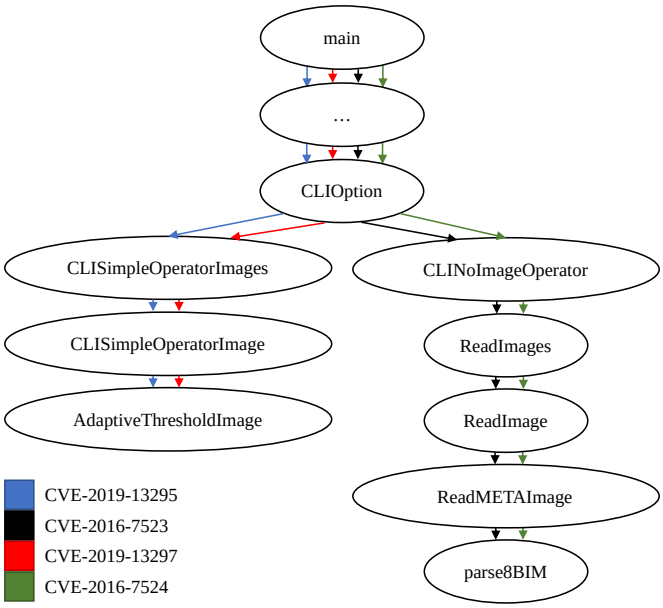

Figure 6: Relationship among four Common Vulnerability and Exposures assigned for ImageMagick over a three year period.

1) **Insights and lessons learned:**

Given how easily we found the duplicates, we were surprised that these bugs were not marked as duplicates in the vulnerability databases [34, 37, 3]. To gain insights into why multiple CVEs were assigned for the same bug, we closely examined these vulnerability databases and bug trackers. Our findings reveal several common causes for duplicates: (*i*) developers being unaware of CVE requests, (*ii*) testers opening separate issues for each test case that triggers the same bug (*iii*) the same bug being triggered in a different function or line number on a different program version, (*iv*) multiple testers submitting the same bug at different points in time and (*v*) incorrect fixes. Table V lists all the duplicate CVEs we discovered for `ImageMagick` and `JasPer` and the methods we used to verify the duplicates. We also discovered 15 duplicates for `FFmpeg`, 4 for `livarchive`, 2 for `libxml2`, 17 for `PHP` and 5 for `Wireshark`.

These findings aptly show that previously known bugs can, and will, appear again and again in later versions of a program, albeit with slightly varied fingerprints [43]. Based on our experiences, we recommend a common data format (see Figure 7) for identifying and analyzing the relationships among duplicates and inexact duplicates. The code and datastore are available[3]. Entries in the datastore abide by the format below.

```
{
    "query_id": "CVE-2016-7515",
    "similarity_groups": [{
            "bug_id": "CVE-2016-7519",
            "similarity": 1.0
        },
        {
            "bug_id": "CVE-2017-6500",
            "similarity": 0.7265
        }
    ],
    "analysis_groups": [{
        "group_id": "7789545",
        "bugs": [
            "CVE-2016-7515", "CVE-2016-7519", "CVE-2017-6500"
        ],
        "sequence": "main, MagickMain, MagickCommandGenesis,
        MagickImageCommand, ProcessCommandOptions, CLIOption,
        CLINoImageOperator, ReadImages, ReadImage",
        "shared_length": "9 of 10"
    }]
}
```

Figure 7: Example format.

D. RQ4: Can CrashSearch effectively reduce the number of crashes triagers have to analyze?

To evaluate how well CrashSearch can be used to reduce the number of crash reports generated by automated testing, we simulated a real-world scenario where four independent groups submit different sets of crash reports for the same set of programs. We use a subset of the programs used in §V-A: `libarchive`, `libjpeg-turbo`, `libtiff`, and `libxml2`. Our fuzzers include `AFL`, `Angora`, `FairFuzz`, and `MOpt`. We choose these four fuzzers because of their popularity and the significance of the differences among their performance [25]. Our selection reflect different points in the fuzzing space — for example, MOpt offers a unique mutational scheduling scheme and FairFuzz applies a novel seed selection procedure that attempts to guide the fuzzer toward rarely executed paths. We only submit the unique crashes by each fuzzer, as determined by its built-in deduplication processes. In total, there are 1055 crashes corresponding to 13 unique bugs.

We assume that the triager receives the crash reports found by the corresponding groups in independent batches. The triager then runs CrashSearch as part of their workflow. In the beginning, the triager has an empty datastore and uses CrashSearch to initialize it with the unique fingerprints from

[3]https://github.com/kedjames/crashsearch-triage

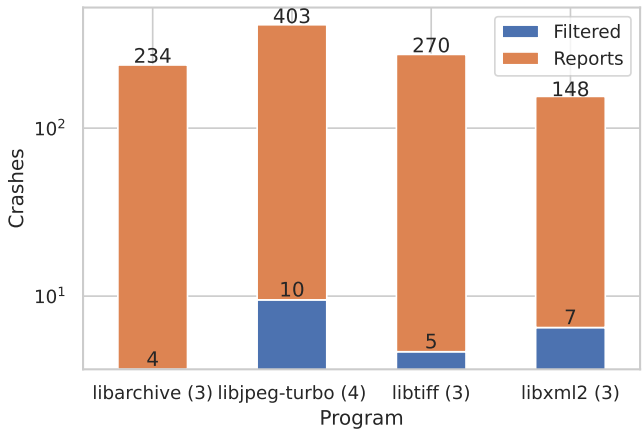

Figure 8: The number of crashes generated by four fuzzers and the average number of crashes after filtering with CrashSearch. The numbers in the parenthesis after the program names represent the number of unique bugs for such program associated with the crashes.

the first batch of crash reports. The datastore is updated after each batch of reports are processed by CrashSearch. We perform this experiment 24 times to cover all possible inter-arrivals of bug reports by independent groups. Figure 8 shows the results. Our results show that CrashSearch reduces the number of crash reports received per project from 263 on average to 10 or less unique fingerprints. On average, after CrashSearch filters the crash reports, each unique bug in a program contains only two crash reports. More importantly, regardless of the coverage of fingerprints in the initial datastore, in the end, there was at least one fingerprint for each of the bugs in the ground truth.

a) **Insights and lessons learned:**

Despite (or maybe in spite of) the fact that we choose popular graybox fuzz testing suites, each fuzzer generated tens of unique crash reports in a single day, but the vast majority of these crashes were instances of the same flaws. This finding corroborates that of Wang et al. [49] that shows that no single fuzz testing technique significantly outperforms the others. Even at a conservative estimate of 0.5 man hours per report (*i.e.,* well below the findings of Anvik et al. [5]), that would relate to about 2 weeks of work per project, on average. With multiple independent groups running their fuzzing campaigns for longer periods of time than we did, the deluge of duplicates would be unbearable (taking more than 3 months in total). Indeed, if the same developers were responsible for the 4 libraries listed above, the triager(s) would need to sift through over a thousand crash reports to find the handful of bugs within. *Armed with CrashSearch, however, such winnowing would be greatly reduced as the triager would only have to inspect less than 1/40 of the reports*. Under the same conservative assumption of 0.5 man hours per report, even if the same developers were responsible for all 4 libraries, a small team of 2 developers would now be enough to analyze the crashes within one day.

VI. Limitations

Since we use features including the signal and the crashing line, CrashSearch supports only bugs that result in a crash. For other bug types, we could potentially use different set of features to generate bug fingerprints, but we leave the exploration as future work.

For our evaluation, the main threat of validity is the size of our dataset and how representative it is. While our dataset includes a limited number of unique bugs, our dataset includes popular programs in a variety of categories, and we followed the same procedure to generate duplicate fingerprints as prior studies [45, 9]. Therefore, we believe that our findings regarding CrashSearch should apply to real-world fuzzing campaigns.

VII. Conclusion

We present an end-to-end design for identifying instances of known bugs given the symptoms of a crash. Our solution utilizes a novel combination of the MinHash algorithm and Locality Sensitive Hashing. We leverage MinHash to efficiently estimate the Jaccard similarity of features, and we utilize Locality Sensitive Hashing during the indexing phase to eliminate the need for pairwise comparisons. We evaluated the effectiveness of our approach and showed that it is both more accurate and faster than state-of-the-art approaches for stack backtrace similarity detection. More importantly, our investigations demonstrate that the approach helps simplify the work flow for an analyst inspecting crashes during triage, and help find relationships among crashes that may not be readily obvious.

VIII. Acknowledgements

We thank the reviewers for their valuable feedback. We also express our gratitude to Ryan Court, Kevin Z. Snow, and Kevin Valakuzhy for their support and insightful discussions throughout the course of this project.

References

[1] Bug writing guidelines. URL https://bugzilla.mozilla.org/page.cgi?id=bug-writing.html.

[2] A null pointer dereference. URL https://github.com/jasper-software/jasper/issues/269.

[3] Serkan Özkan. Cve details, 2012. URL https://www.cvedetails.com/.

[4] S. Alrabaee, M. Debbabi, P. Shirani, L. Wang, A. Youssef, A. Rahimian, L. Nouh, D. Mouheb, H. Huang, and A. Hanna. *Binary Code Fingerprinting for Cybersecurity*. 2020.

[5] J. Anvik, L. Hiew, and G. C. Murphy. Coping with an

open bug repository. In *OOPSLA Workshop on Eclipse Technology EXchange*, pages 35–39, 2005.

[6] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Intl. Conference on Software Engineering*, pages 361–370, 2006.

[7] K. Bartz, J. W. Stokes, J. Platt, R. Kivett, D. Grant, S. Calinoiu, and G. Loihile. Finding similar failures using callstack similarity. 2008.

[8] N. Bettenburg, R. Premraj, T. Zimmermann, and K. Sunghun. Duplicate bug reports considered harmful ... really? In *Intl. Conference on Software Maintenance*, pages 337–345, 2008.

[9] T. Blazytko, M. Schlögel, C. Aschermann, A. Abbasi, J. Frank, S. Wörner, and T. Holz. {AURORA}: Statistical crash analysis for automated root cause explanation. In *USENIX Security Symposium*, pages 235–252, 2020.

[10] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury. Directed greybox fuzzing. In *ACM Conference on Computer and Communications Security*, pages 2329–2344, 2017.

[11] A. Z. Broder. On the resemblance and containment of documents. In *Compression and Complexity of SEQUENCES 1997*, pages 21–29, 1997.

[12] M. Brodie, S. Ma, G. Lohman, L. Mignet, N. Modani, M. Wilding, J. Champlin, and P. Sohn. Quickly finding known software problems via automated symptom matching. In *Conference on Autonomic Computing*, pages 101–110, 2005.

[13] M. Brodie, S. Ma, L. Rachevsky, and J. Champlin. Automated problem determination using call-stack matching. *Journal of Network and Systems Management*, 2005.

[14] J. C. Campbell, E. A. Santos, and A. Hindle. The unreasonable effectiveness of traditional information retrieval in crash report deduplication. In *ACM Working Conference on Mining Software Repositories*, pages 269–280, 2016.

[15] M. Castelluccio, C. Sansone, L. Verdoliva, and G. Poggi. Automatically analyzing groups of crashes for finding correlations. In *Foundations of Software Engineering*, pages 717–726, 2017.

[16] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *ACM Symposium on Theory of Computing*, pages 380–388, 2002.

[17] W. Cui, M. Peinado, S. K. Cha, Y. Fratantonio, and V. P. Kemerlis. RETracer: Triaging crashes by reverse execution from partial memory dumps. In *Intl. Conference on Software Engineering*, 2016.

[18] W. Cui, X. Ge, B. Kasikci, B. Niu, U. Sharma, R. Wang, and I. Yun. REPT: Reverse debugging of failures in deployed software. In *USENIX Conference on Operating Systems Design and Implementation*, pages 17–32, 2018.

[19] 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 *Intl. Conference on Software Engineering*, pages 1084–1093, 2012.

[20] S. Das, K. James, J. Werner, M. Antonakakis, M. Polychronakis, and F. Monrose. A flexible framework for expediting bug finding by leveraging past misbehavior to discover new bugs. In *Annual Computer Security Applications Conference*, 2020.

[21] T. Dhaliwal, F. Khomh, and Y. Zou. Classifying field crash reports for fixing bugs: A case study of mozilla firefox. pages 333–342, 2011.

[22] J. Foote. Exploitable crash analyzer version 1.6, 2015. URL https://github.com/jfoote/exploitable.

[23] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Intl. Conference on Very Large Databases*, page 518–529, 1999.

[24] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: Ten years of implementation and experience. In *ACM Symposium on Operating Systems Principles*, pages 103–116, 2009.

[25] Google. Fuzzbench: 2020-04-21 report. 2020.

[26] J. He, L. Xu, M. Yan, X. Xia, and Y. Lei. Duplicate bug report detection using dual-channel convolutional neural networks. In *Intl. Conference on Program Comprehension*, pages 117–127, 2020.

[27] G. Jeong, S. Kim, and T. Zimmermann. Improving bug triage with bug tossing graphs. In *ACM Foundations of Software Engineering*, pages 111–120, 2009.

[28] A. Khvorov, R. Vasiliev, G. Chernishev, I. M. Rodrigues, D. Koznov, and N. Povarov. S3m: Siamese stack (trace) similarity measure. *arXiv preprint arXiv:2103.10526*, 2021.

[29] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating fuzz testing. In *ACM Conference on Computer and Communications Security*, pages 2123–2138, 2018.

[30] A. Kukkar, R. Mohana, Y. Kumar, A. Nayyar, M. Bilal, and K.-S. Kwak. Duplicate bug report detection and classification system based on deep learning technique. *IEEE Access*, 8, 2020.

[31] J. Lerch and M. Mezini. Finding duplicates of your yet unwritten bug report. In *European Conference on Software Maintenance and Reengineering*, pages 69–78, 2013.

[32] J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of Massive Datasets*. 2nd edition, 2014.

[33] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo. The art, science, and engineering of fuzzing: A survey. 2019.

[34] MITRE. Common vulnerabilities and exposures (cve), 1999. URL https://cve.mitre.org/.

[35] N. Modani, R. Gupta, G. Lohman, T. Syeda-Mahmood, and L. Mignet. Automatically identifying known soft-

ware problems. In *Intl. Conference on Data Engineering Workshop*, pages 433–441, 2007.

[36] Mozilla. Mozilla rr, 2018. URL https://rr-project.org/.

[37] NIST. National vulnerability database, 2000. URL https://nvd.nist.gov/.

[38] L. Poddar, L. Neves, W. Brendel, L. Marujo, S. Tulyakov, and P. Karuturi. Train one get one free: Partially supervised neural network for bug report duplicate detection and clustering. *arXiv preprint arXiv:1903.12431*, 2019.

[39] E. Raff and C. Nicholas. Malware classification and class imbalance via stochastic hashed lzjd. In *ACM Workshop on Artificial Intelligence and Security*, pages 111–120, 2017.

[40] I. M. Rodrigues, D. Aloise, and E. R. Fernandes. Fast: A linear time stack trace alignment heuristic for crash report deduplication. In *Intl. Conference on Mining Software Repositories*, pages 549–560, 2022.

[41] K. K. Sabor, A. Hamou-Lhadj, and A. Larsson. Durfex: a feature extraction technique for efficient detection of duplicate bug reports. In *IEEE Intl. Conference on Software Quality, Reliability and Security*, pages 240–250, 2017.

[42] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

[43] M. Stone. The state of 0-day in-the-wild exploitation. USENIX Association, Feb. 2021.

[44] J. Uddin, R. Ghazali, M. M. Deris, R. Naseem, and H. Shah. A survey on bug prioritization. *Artificial Intelligence Review*, pages 145–180, 2016.

[45] R. van Tonder, J. Kotheimer, and C. Le Goues. Semantic crash bucketing. In *ACM/IEEE Intl. Conference on Automated Software Engineering*, pages 612–622, 2018.

[46] R. Vasiliev, D. Koznov, G. Chernishev, A. Khvorov, D. Luciv, and N. Povarov. Tracesim: a method for calculating stack trace similarity. In *ACM Intl. Workshop on Machine-Learning Techniques for Software-Quality Evaluation*, pages 25–30, 2020.

[47] D. Vyukov. Syzbot and the tale of thousand kernel bugs. Aug. 2018.

[48] J. Wang and J. Han. Bide: Efficient mining of frequent closed sequences. In *Intl. Conference on Data Engineering*, pages 79–90. IEEE, 2004.

[49] J. Wang, Y. Duan, W. Song, H. Yin, and C. Song. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *Symposium on Recent Advances in Attacks and Defenses*, pages 1–15, 2019.

[50] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *ACM/IEEE Intl. Conference on Software Engineering*, pages 461–470, 2008.

[51] Q. Xie, Z. Wen, J. Zhu, C. Gao, and Z. Zheng. Detecting duplicate bug reports with convolutional neural networks. In *Asia-Pacific Software Engineering Conference*, pages 416–425. IEEE, 2018.

[52] J. Xu, D. Mu, X. Xing, P. Liu, P. Chen, and B. Mao. Pomp: Postmortem program analysis with hardware-enhanced post-crash artifacts. In *USENIX Security Symposium*, pages 17–32, 2017.

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

[54] Zeropoint Dynamics. Zelos crashd plugin, 2020. URL https://github.com/zeropointdynamics/zelos-crashd.