Directing a Search Towards Execution Properties with a Learned Fitness Function

Leonid Joffe and David Clark
{*leonid.joffe.14, david.clark*}*@ucl.ac.uk*

University College London

***Abstract*—Search based software testing is a popular and successful approach both in academia and industry. SBST methods typically aim to increase coverage whereas searching for executions with specific properties is largely unresearched. Fitness functions for execution properties often possess search landscapes that are difficult or intractable. We demonstrate how machine learning techniques can convert a property that is not searchable, in this case crashes, into one that is. Through experimentation on 6000 C programs drawn from the Codeflaws repository, we demonstrate a strong, program independent correlation between crashing executions and library function call patterns within those executions as discovered by a neural net. We then exploit the correlation to produce a searchable fitness landscape to modify American Fuzzy Lop, a widely used fuzz testing tool. On a test set of previously unseen programs drawn from Codeflaws, a search strategy based on a crash targeting fitness function outperformed a baseline in 80.1% of cases. The experiments were then repeated on three real world programs: the VLC media player, and the libjpeg and mpg321 libraries. The correlation between library call traces and crashes generalises as indicated by ROC AUC scores of 0.91, 0.88 and 0.61. The produced search landscape however is not convenient due to plateaus. This is likely because these programs do not use standard C libraries as often as do those in Codeflaws. This limitation can be overcome by considering a more powerful observation domain and a broader training corpus in future work. Despite limited generalisability of the experimental setup, this research opens new possibilities in the intersection of machine learning, fitness functions, and search based testing in general.**

***Index Terms*—Search based software engineering, Search based software testing, Fuzzing, Machine learning**

I. Introduction

A. Motivation

Testing is an integral part of the software engineering process. It has been estimated to account for over 50% of development costs [1]. Its purpose is to find faults in software by executing a program [2]. Search Based Software Testing (SBST) is a popular technique in testing. Any SBST approach depends on two choices: a *representation* and a *fitness function* [3]. A representation is what the tester chooses to observe about a program and its execution, and an *observation* is an instance of a representation that has been captured. A fitness function scores the intermediate candidate solutions to direct the search. A typical fitness function in SBST aims to improve coverage [4].

Searching to optimise coverage criteria has attracted significant research effort, though coverage has been observed to have limitations [5], [6], [7], [8], [9]. Fitness functions targeting executions with particular properties, by contrast, is a relatively unexplored area. It is not immediately obvious how to go about creating an effective, targeted, fitness function for an arbitrary property. What information to observe? How to interpret the representation as a fitness landscape?

B. Overview

This paper aims to show how Machine Learning (ML) can be used to create a useful fitness function in the context of searching for executions with a specific property. We first show that ML can be used to process observations to identify executions with a particular property. We then show how this capability can be used to guide an SBST process to target the property. The hypothesis behind the proposed approach is that a classifier's likelihood estimate of the presence of a crash can be interpreted as a fitness – a candidate solution's proximity to a crash. Prioritising candidate solutions that are "closer" to a crash ought to lead to more efficient crash discovery.

We present four central findings. First, we show that the C library call traces have a strong correlation with the presence of a crash. Second, we show that a fitness function based on the crash likelihood has a clear guiding effect on the search strategy *vs* a baseline. Third, we show that the correlation for a presence of a crash generalises to real world programs. Finally, we present a mixed result for the generalisability of the fitness function driven by a crash likelihood: the fitness landscape is rich for some programs and the targeted search works, but for other programs the representation of standard C library call traces is inadequate.

C. Approach

The proposed approach has two main steps. First, an ML mechanism is trained to predict the presence of a property based on observations of a program's execution. Then, the prediction of the property is used as a basis for a fitness function in an SBST process.

We demonstrate the methodology by considering standard C library call traces as the representation, and crashes as the execution property of interest. First we collect standard C library call traces and crash / no crash labels across a corpus of programs. We then train a neural network classifier on a corpus of execution traces of standard C library calls with crash labels.

We then couple the crash likelihood classifier with the American Fuzzy Lop (AFL) fuzzer [10]. AFL relies on a number of heuristics to guide its search. We introduce the crash likelihood as an additional prioritisation heuristic: *non-crashing* executions to which the classifier assigns a higher crash likelihood are given a higher priority during fuzzing.

D. Contributions

The main contributions of this work are (a) a novel approach to constructing a fitness function for an SBST process and (b) a case study demonstrating that this works in practice. The presented methodology produces a meaningful fitness function from an arbitrary representation. The approach requires neither access to source code nor analytical processing of observations. The methodology opens new research possibilities for the use of ML in constructing effective fitness functions for SBST. In future work, it will be extended to other observations and execution properties.

II. Research Goals

There are three research goals. The first investigates the validity of the chosen representation with respect to the execution property of interest. The second looks at the use of an ML constructed fitness function based on the representation. The third addresses the generalisability of the method to real world applications.

A. Representation

For a representation to be useful as a basis of a fitness function, it needs to be correlated with the execution property of interest. We need to determine whether program executions represented with standard C library calls are correlated with crashes, when interpreted by a neural network. The first research question of this paper is **RQ1**:
"Are traces of standard C library calls correlated with crashes?"

B. Fitness Function

Once we have a representation, we use it to construct a fitness function. The effectiveness of the fitness function then needs to be evaluated. This is done by comparing its crash discovery rate against a baseline – an SBST process guided by a non-targeted fitness function. The second research question is **RQ2**:
"Does the property targeting fitness function result in a higher crash discovery rate than a baseline?"

C. Generalisability to real world Programs

Effectiveness of machine learning techniques depends on the representativeness of the training data. This paper presents a proof of concept, and the generalisation of the trained model to arbitrary real world programs is expected to be limited. The following two research questions investigate the degree to which the model generalises.
RQ3.1:*"What is the crash predicting ability of a pre-trained model on real world applications?"*
RQ3.2:*"Is a fitness landscape produced by a pre-trained model applicable to real world applications?"*

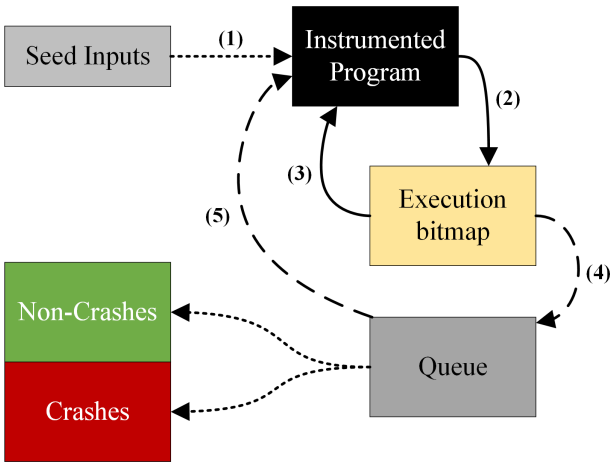

Fig. 1: AFL workflow. Solid arrows indicate inner loop, dashed arrows outer loop, and dotted lines initialisation and termination steps. Initial seeds are provided by the user **(1)**. The instrumented program is executed with a queue element and the trace is recorded **(2)**. The element is then modified and the program re-executed **(3)**. Traces deemed interesting are appended to the queue **(4)**. Once a cycle completes, all queue elements are scored **(5)** and a new cycle begins. At termination, the queue and unique crashes remain available.

III. Background

Our work relies on the American Fuzzy Lop (AFL) fuzzer as the underlying SBST tool and Valgrind as the program profiling tool. We present their functionality below. Machine learning concepts used by our classifier are also introduced.

A. American Fuzzy Lop

AFL is a popular diversity driven fuzzer, whose main purpose is to discover crashes in programs. At its core, it implements a diversification strategy on observed program traces, based on several heuristics. Its basic workflow can be seen in Fig. 1, and a detailed description is presented below.

1) Initialisation: The user provides one or more inputs as seeds which forms the initial queue. Each queue element is calibrated: it is executed, and its running time and execution trace are recorded.

2) Fuzzing Operations: Each queue item is then modified and executed repeatedly for a number of executions determined by its *fuzzing budget*. These include deterministic and stochastic modifications. The deterministic steps are bit and byte flips, simple arithmetic and "interesting integers". The stochastic steps are combinations of the above, block deletions, insertions and overwrites, memset and splicing. Further details of these operations can be found online [11].

3) Trace representation: Before fuzzing, AFL instruments a program at every decision point. The transitions between these points form a hashmap (in AFL's documentation this is called a "bitmap") of edges and their hit counts. To fit the bitmap fully into L2 cache, only a rough number of transition counts is kept: 0, 1, 2, 3, 4–7, 8–15, 16–31, 32–127, 128+. In SBST terminology, the bitmap is AFL's representation.

4) Interestingness: AFL defines a notion of interestingness (used interchangeably with uniqueness) of an execution on

the state transition bitmap by one of these three conditions: the execution produced a state transition that has not been seen before; it changed the count of a transition; or it did not exercise a transition which had previously always been exercised. This definition of interestingness is heuristically chosen as a trade-off between the efficiency of crash discovery and instrumentation overhead. For an input to be appended to the queue, its execution trace needs to fulfil one of these conditions. Crashing inputs are *not* used as seeds for further fuzzing by default.

5) Prioritisation Heuristics: A single pass through the queue is called a cycle. At the end of a cycle, old queue elements whose bitmaps are entirely superseded by new ones are culled. Fuzzing then continues with an updated queue. At the start of a new cycle, each queue element is assigned a fuzzing budget α. The value determines how many times each element is to be modified and executed. This is AFL's *fitness function*. There are five heuristics.

- An element's execution time with respect to the average. Quicker executions get a higher priority simply to allow for more total executions.
- The size of the element's bitmap. The reasoning is that larger traces are more likely to exercise more behaviours and thus reveal more faults.
- Recency of a queue element. Newer entries are prioritised as they would have gotten less air time than older ones.
- An element's hierarchic novelty. For instance, the depth of entries that were found using the initial queue is 1. Elements that were found by modifying depth 1 elements would have a depth of 2 and so on. Elements of greater depth (i.e. descendants of more complex parents) get a higher priority as they are expected to produce ever more complex executions.
- Ad-hoc budget extension. If unique executions are found while fuzzing an element, its budget is dynamically increased until no more new ones are found.

B. Valgrind

Valgrind is an instrumentation framework for dynamic analysis which has been used extensively in academia and industry [12], [13]. It tracks every instruction as a program is executed in a simulated environment. One of the tools included is Callgrind, which records an execution's call history [14]. The calls include both the program's own functions as well as any library calls it makes. Traces produced by Callgrind are ideal for our purposes for two reasons. First, we expect function calls to be strongly correlated with crashes. Second, since library calls can be isolated from a program's own function calls, these traces can be compared across programs.

C. Machine Learning Mechanisms

The structure of a neural network may vary greatly depending on the intended purpose and data. The central features of our NN are outlined next.

Batch normalisation is a method for standardising data within the network so that it has a mean of 0.0 and standard deviation of 1.0 [15]. Not only does this make the training faster, but it also rids us of the need for preliminary data normalisation.

An activation function of a neuron determines the range of its outputs. A hyperbolic tangent (tanh) activation outputs values in $(-1.0, 1.0)$ whilst sigmoid activation produces values in $(0.0, 1.0)$. The former is used to introduce non-linearity for the network to capture complex, non-linear relationships in the data. The latter is used to produce a likelihood estimate for binary classification problems.

A typical strong binary classifier produces output values close to zero or one, indicating the value of a binary label. This is not ideal for our purpose, since we want a variety of degrees of certainty where a search can be conducted. Adding Gaussian noise regularisation is one way of introducing uncertainty into the model [16]. In addition to producing a wider range of certainties, it also makes the model more robust against overfitting.

D. Receiver Operator Characteristic

The Receiver Operator Characteristic is a mechanism to measure the effectiveness of a binary classifier. It is a plot of the true positive rate against the false positive rate. The Area Under Curve (AUC) of the ROC is a summary of the ROC and we use it our evaluation. The main benefit of ROC AUC versus accuracy is that it is independent of the balance of labels [17].

For instance, if the proportion of class one and class zero in a test dataset is 1:9, a classifier would achieve an accuracy of 0.9 by always guessing zero. Since AUC of ROC is the *rate* of true positives, the class sizes are normalised, and a classifier guessing the larger labels achieves an AUC of 0.5. This is a more honest representation of its performance.

IV. Experimental Setup

This section describes our experimental setup. It consists of a large dataset of instrumented program executions, a neural network regression classifier and a modified AFL fuzzer. In terms of the implementation, our contribution is the additional prioritisation heuristic to AFL – one generated by the NN (see Subsection IV-F below).

A. Augmented Codeflaws

The main corpus used in this work is extracted from the CodeFlaws program repository. It contains a total of 7808 programs with sizes ranging from 1 to 322 lines of code, along with test cases and automatic fix scripts [18]. These are grouped into 40 defect classes, with triggering inputs and automatic repair scripts. As it is, there are two drawbacks to this dataset. First, there are too few inputs to train an Neural Network (NN) model on. An appropriate dataset would contain tens of thousands of program executions. Second, all of the programs are perfectly valid C code, and the defects of the faulty versions are relative to the correct program. That is, the defective programs are only faulty with respect to the fixed programs, i.e. oracles. In this paper, we bypass the oracle

problem [19] and concentrate on a universally undesirable property – a crash.

The dataset was augmented by fuzzing the programs in Codeflaws. Each program was fuzzed with AFL using default settings for five minutes, using six parallel instances of AFL. Due to AFL's generation heuristics, some programs ended up with a disproportionately large number of inputs. Of these, we uniformly sampled a subset of a thousand executions to avoid skewing the training data towards a particular program.

The dataset was divided into a train set and a test set. To ensure that the test dataset was not tainted with training data, the train-test partition was created as follows. The programs in the Codeflaws repository are grouped pairwise by defect IDs. That is, each element in the repository is a pair consisting of a faulty and a fixed version of a program. The difference between a fixed and a faulty program tends to be small, e.g. change of one line. Having one program from a pair in the training set and the other in the test set would overfit the data: the model would get trained on an example which is very similar to one in the test set. Thus we need to ensure that only one program in a fixed-faulty pair is included in the training set. The train-test split was therefore done on a granularity of defect IDs. For instance, there are seven pairs of programs with a 38-B defect ID. All seven pairs of the 38-B defect would be excluded from the training set and kept for testing. This ensured that the programs in the test set are sufficiently different from the ones in the training set. A 75%:25% train-test split was produced by uniformly sampling the defect IDs.

B. Real World Programs

We consider three real world programs to investigate the generalisability of a Codeflaws trained model. They are the *VLC* 2.0.3 media player, and the *libjpeg* and *mpg321* image and mp3 libraries. The version of VLC was chosen because it was listed on AFL's site and known to have a crash causing fault in its subtitle module [10]. The latter two are also known to have crash producing faults, and have been used as benchmarks in recent work on fuzzing (see Section VI). The initial seed input for each program was a blank file.

C. Trace Instrumentation

The dataset was then instrumented with Valgrind's Callgrind tool which produces a trace of function calls. An example can be seen in Listing 1. Traces were then parsed into call counts of the number of times each function called another. Program specific functions were filtered out, so each execution is only characterised by its trace of standard C library functions. This preprocessing produced a corpus of instrumented executions across a range of various programs, with crash / no crash labels.

```
< .../dl-runtime.c:_dl_fixup (4x) \\A
< .../dl-machine.h:_dl_relocate_object (80x) \\B
* .../dl-lookup.c:_dl_lookup_symbol_x \\C
> .../dl-lookup.c:do_lookup_x (84x) \\D
```

Listing 1: Example of a Valgrind instrumented trace. The symbols <, * and > correspond to "called by", "function" and "calls" respectively. The last number in braces shows how many times this call occurred in a given execution. In this example function on row C is called by functions on rows A and B, 4 and 80 times respectively, whilst it calls function on row D 84 times. The trace produced is {(A, C : 4), (B, C : 80), (C, D : 84)}

	Mean	Median	Stdev
Crashes per program	8.886	1	42.730
Execs per program	61.192	25	127.535
Crashes per program	0.153	0.0309	0.309
Test set size	1537.33	1524	82.59

TABLE I: General characteristics of the corpus

D. Experiments

For RQ1, the training and testing procedure was repeated 20 times, and the results averaged. The characteristics of the produced corpus can be seen in Table I. The corpus had a total of 6089 programs.

For RQ2, using the whole test set of some 1500 programs would not have been feasible. Therefore, 200 programs were sampled, uniformly at random, from the test set. The only restriction in selecting the programs was that they showed at least a single crash in the corpus generation stage (see Section IV-A). Each program was then fuzzed for up to two times the number of executions in the corpus generation stage or to a maximum of 15 minutes. The total number of crashing executions, the number of unique crashes and the number of distinct paths was recorded.

For RQ3, the NN was *not* retrained or modified. Due to time constraints, experiments were not carried out on all modes – only not B4 and T4 (see Subsection IV-F below). They are the most representative benchmark between a "blind" strategy and a guided one; effects of search strategies are not impacted by AFL's built-in heuristics. Each mode was run for 12 hours, five times each.

E. Neural Network Classifier

A regression classifier is a model that outputs a floating point likelihood of an input belonging or not belonging to a class. The output can be thought of as a probability of the input being part of a class or not. Our classifier is based on a neural network. An NN was chosen because it does not require feature extraction pre-processing and it can be easily modified to use alternative data inputs. The NN takes instrumented traces of program executions as inputs and a binary label of a crash as the target. The NN's output is the likelihood of a crash for a given execution trace. The framework related to RQ1 is pictured in Fig. 2.

The network is composed of seven layers. The input layer takes the vector of function call counts. There were 1733 distinct function call transitions in the corpus, which is therefore the size of the input layer. The second layer is batch normalisation. Layers three through six are densely connected

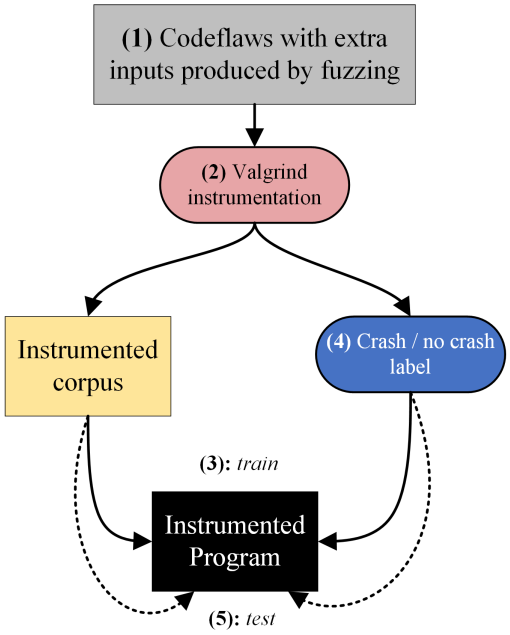

Fig. 2: Experimental setup for RQ1. Codeflaws is a augmented with additional inputs producing crashing and non-crashing executions **(1)**. These are then instrumented with Valgrind **(2)**. A neural network is trained on the instrumented executions **(3)** and crash / no crash labels **(4)**. The performance of the NN classifier is then evaluated on a held out test set **(5)**.

.

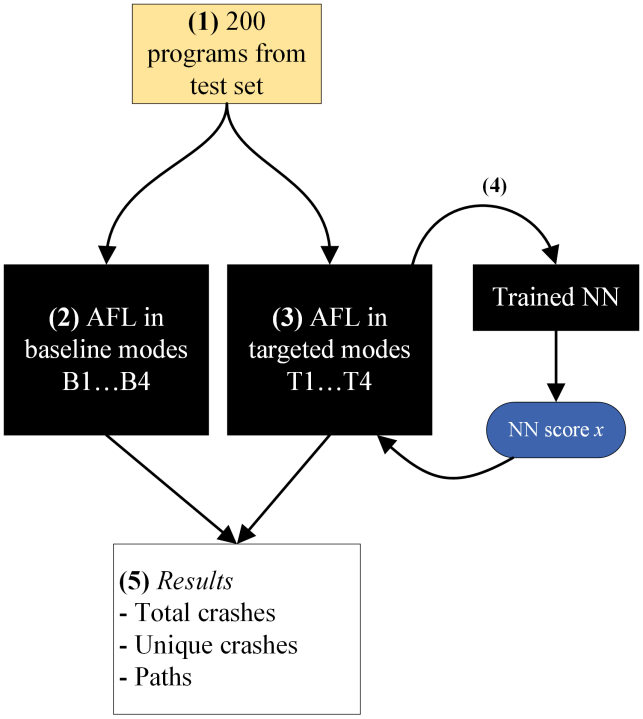

Fig. 3: Experimental setup for RQ 2. 200 programs are sampled from the test set **(1)**. The programs are fuzzed in baselines modes **(2)** and modified targeted modes **(3)**. Targeted modes query a trained NN for crash likelihood scores **(4)**. The number of total crashes, unique crashes and discovered paths are recorded **(5)**.

layers with sizes 128, 64 and 32 neurons with Gaussian noise regularisation. Hyperbolic tangent activation is used for non-linearity. The last layer is a single neuron with a sigmoid activation. The network was implemented using the Keras framework and trained using the RMSprop optimiser [20].

F. Constructing the Fitness Function

We modify AFL's fitness function to consider the crash likelihood score provided by the NN. This is done with a number of modifications outlined below. The framework is shown in Fig. 3.

Calls to trained NN for new queue elements: AFL adds an input to its queue when the execution path is found to be interesting, as explained above. A newly added queue element is sent to a mechanism which executes it under Valgrind, instruments the trace and scores it with a trained NN model. We tried doing this analysis with AFL's uniqueness heuristics disabled, i.e. by calling it for each modified input during fuzzing. This was prohibitively slow due to the instrumentation overhead, and the signal from the model was too weak: the crash likelihood scores would rarely vary. This means that the execution paths appear very similar in terms of the standard C library call representation under the slightly mutated inputs. AFL's built-in definition of uniqueness was thus maintained, and the framework only asks for the model's estimate intermittently – at the start of a new cycle.

Normalisation of crash likelihood scores: The crash likelihood scores are then used to calculate an alternative fuzzing budget β (equations 2 and 1). The values produced by the NN are not directly comparable across programs. For instance, executions of program A might have a mean of 0.1, while the values for program B may have a mean of 0.6, i.e. 0.5 is not guaranteed to be the cut-off point of "suspicious" vs "not suspicious". The raw values are thus normalised as shown in Eq. 2. This maps the crash likelihood score into a range of $[0.0, 2.0]$, where scores between x_{min} and μ get an adjusted score $w(x)$ in the range $[0.0, 1.0]$ and those above the mean in $(1.0, 2.0]$.

Fuzzing budget weighting: The normalised score $w(x)$ is a multiplier for the final fuzzing budget β (Eg. 1). Depending on the mode, the initial fuzzing budget α is either a fixed value (e.g. $\alpha = 100$ for T2) or one given by AFL's heuristics. This value is then weighted by the normalised score $w(x)$. In addition, to prevent discarding elements simply due to the NN's opinion, a minimal value is introduced. This assigns adjusted values at least some fuzzing budget, and prevents them from being discarded simply because of being non-suspicious.

$$\beta = \max(20,\ \alpha \times w(x)) \tag{1}$$

$$w(x) = \begin{cases} \frac{x - x_{min}}{\mu - x_{min}}, & \text{if } x \leq \mu \\ 1 + \frac{\mu - x}{x_{max}}, & \text{otherwise} \end{cases} \tag{2}$$

Queue Sorting: AFL uses a first in first out queue. We introduce queue sorting by descending β, which is done at the start of every cycle. A fuzzing cycle therefore starts with the most suspicious seeds. This is important due to AFL's ad-hoc fuzzing budget extension heuristic. Without modification, AFL dynamically increases the assigned budget if the seed element keeps producing new paths. This way it may end up discovering new paths from seeds with low crash likelihood scores.

Disabling Ad-hoc Extension: Without modification, AFL dynamically increases the assigned budget if the seed element keeps producing new paths. This way AFL may end up discovering new paths from seeds regardless of their budget. To isolate the effect of a particular prioritisation strategy, modes 3 and 4 (see below) had the ad-hoc extension disabled. It is an artificial weakening of AFL's heuristics intended for experimental purposes.

Modes of modified AFL: The experiments were run in 8 modes: four baselines (**B1-4**, with fuzzing budget α) and four targeted modes (**T1-4**, with fuzzing budget β). The purpose of several different modified and baseline modes is to investigate whether a targeted strategy noticeably changes the crash discovery rate. Modes other than **B1** and **T1** disable some of AFL's original heuristics in order to make the effect more apparent.

- **B1** The first baseline mode is the unmodified AFL mechanism with ad-hoc extension enabled. The fuzzing budget α is calculated by AFL's original heuristics.
- **B2** The second baseline assigns a constant budget of $\alpha = 100$ to each queue element and uses ad-hoc extension.
- **B3,4** The third and fourth baselines are the same as **B1** and **B2** but with ad-hoc extension disabled.
- **T1** The first modified mode considers both the original AFL weighting and the score given by the NN. The fuzzing budget β is given by weighting the original AFL budget α by the adjusted NN score, and uses ad-hoc extension.
- **T2** The second modified mode assigns the budget β purely from the NN with $\alpha = 100$, and uses ad-hoc extension.
- **T3,4** Third and fourth modified modes are same as **T1** and **T2** but with the ad-hoc extension disabled.

V. Results

This section presents the central results of our experiments. We found that the NN discovers a strong correlation between the chosen representation and the property of interest. We also found that a fitness function based on the NN's crash likelihood estimate is clearly more effective at crash discovery than a baseline. Finally, we observe a mixed result of generalisability of the Codeflaws trained model onto real world applications.

RQ1. Correlation of C Library Call Traces and Crashes

Central Finding: Traces of standard C library calls have a strong correlation with crashing executions. Across the 20 randomly partitioned train-test splits of the Codeflaws corpus, the mean and median area under curve of the ROC are 0.897 and 0.901 respectively. The strong correlation suggests that standard C library calls are a useful representation for crash discovery.

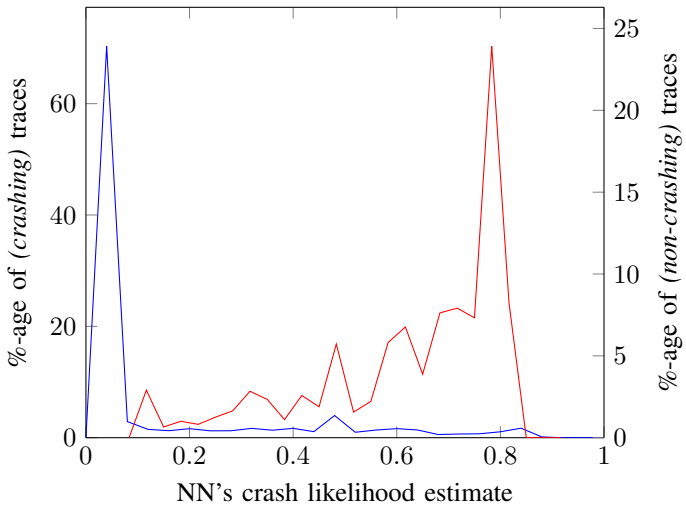

Fig. 4: Distribution of one of the test sets: The x-axis label is the NN's crash likelihood, y-axes (no-crash label on the left, crash label on the right) are the proportions of executions in each 2-percent bucket of x. The two peaks on the left and right indicate the majority of test set elements falling into two distinct classes. Distributions of other sets appeared similar.

The outputs produced by the network are not tightly grouped at the extremes of 0.0 and 1.0. This was observed visually across the score distributions of the test sets. An example of such a distribution is shown in Fig. 4. The ROC curve corresponding to Fig. 4 is presented in Fig. 5. The distribution shows a considerable portion of crashing elements appearing between the two peaks. This space is a middle ground between definitely crashing and definitely not crashing. This distribution of outputs is desirable for the purpose of using the network's output as a fitness function. A distribution with high peaks would have been a search landscape with plateaus – with many candidate solutions sharing the same fitness value, making search impossible. A better accuracy might have been achieved with a stronger classifier if we had not introduced noise to produce this uncertainty. The result is therefore a compromise of the two aims: predictive ability and uncertainty.

RQ2. Fuzzing with a Targeted Fitness Function

Central Finding: Targeted modes have a higher crash discovery efficiency than the baselines. This suggests that the targeted fitness function constructed in this way is effective. The scope of this paper is to show a proof of concept for future work to be built on – not to produce a practically applicable tool. There are recent major improvements on AFL (see Section VI) that the proposed approach does not attempt to outperform.

Data Collected: Three values were recorded for each execution: total number of crashes, number of unique (by AFL's definition of uniqueness) crashes and the number of paths discovered. Fuzzing is inherently random, so each

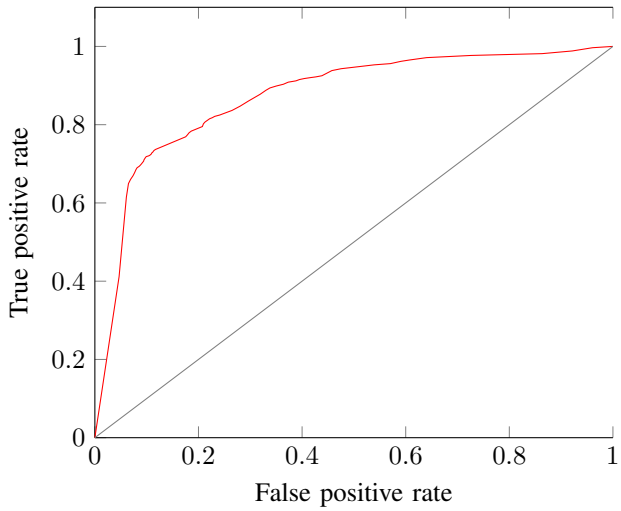

Fig. 5: The Receiver Operator Characteristic curve for the distribution in Fig. 4. The red curve represents the false positive *vs* true positive rate of crashing execution classification, corresponding to the red (RH peak) distribution in Fig. 4. The total AOC here is 0.887. The black diagonal line represents the indifference curve.

program was fuzzed under each experimental mode (B1-4 and T1-4) three times, and the median values were taken. More repetitions would have made the results more robust and convincing but only three runs were taken due to time constraints.

The performance is evaluated with the area under curve for each measure. The reason for accumulating the values in this way is to reward both the number of crashes (or paths) found, as well as finding them sooner. This is illustrated in a plot of the result of fuzzing program 48-C-255529, in Fig. 6. The top plot shows all modes finding 4 or 5 unique crashes, which would suggest equal performance. The bottom plot is the corresponding area under curve of the top right plot. It distinguishes between the modes' performance as some of them found unique crashes quicker than others.

Absolute values are not comparable across different programs: e.g. some programs had one or two unique crashes, others had dozens. Therefore the different modes are ranked based on the cumulative performance measure. The rank yields a score between 0 and 7, for worst and best respectively. The significance of differences between modes was measured by the p-values of pairwise Wilcoxon tests.

We observed that the performance of different modes was indistinguishable for some programs. This was the case for programs with high crash rates and very few paths. If a program constantly crashes, it makes no difference how to look for crashes. Likewise, when a program only has a handful of possible paths, and they all get discovered quickly, the search strategy has no effect. In order to observe a difference in the modes, these trivial programs were filtered out. These included programs with crash rates over 2% and with fewer than 50 paths, leaving a total of 52 programs.

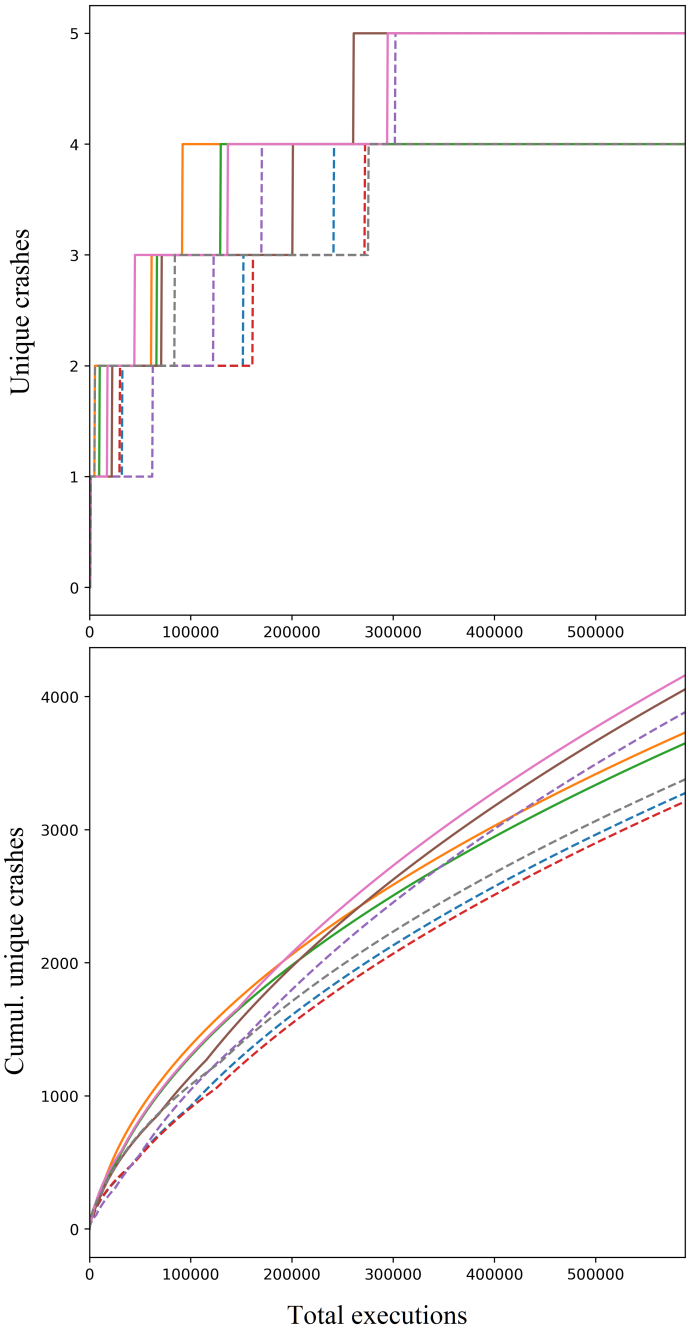

Fig. 6: This figure is an example of performance of the different modes for fuzzing a SUT (48-C-255529 in Codeflaws), showing how our evaluation measures are calculated. The different colours correspond to different modes, with solid lines being the modified modes and the dashed lines being baselines. The x-axis is the number of executions, and the y-axis is the number of unique crashes after x number of executions. The bottom plot shows the same value but accumulated over executions up to that point. A score between 0 and 7 is given to each mode on the cumulative performance.

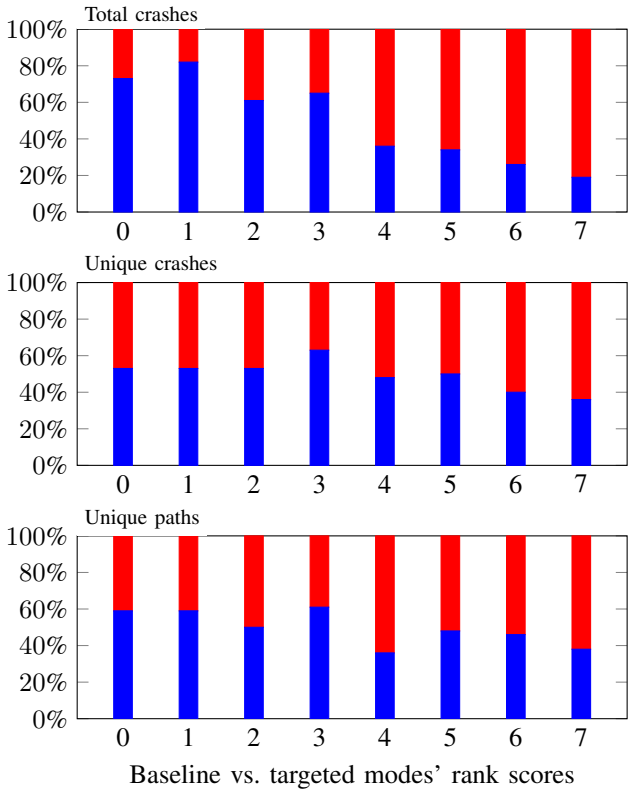

Fig. 7: Distribution of mean scores of baseline modes (blue, bottom part of bar) and targeted modes (red, top part of bar). The plots are of total crashes found, unique crashes found and paths found from top to bottom.

Performance of Targeted Search vs Baseline: The modified modes of AFL produced consistently more crashes than the baseline modes. The scores of baseline modes and targeted modes is shown in Fig. 7, for the 52 non-trivial programs in the test set. The figure shows the proportion of ranks averaged across baseline modes and targeted modes. For instance, the two rightmost columns in the top chart show that a targeted mode was the top performer with a score of 7 for total crashes found in 81% of cases. The effect of a targeted strategy is most apparent in the total crashes found, as evidenced by the top plot. In terms of unique crashes and paths discovered, a targeted mode comes out on top in 63.5% and 61.5% of cases respectively. These findings indicate that an estimate of an NN model trained on standard C library calls of crashing and non-crashing executions has a clear guiding effect.

Performance by Mode: Table II shows a mode-wise comparison of scores. The first column shows that the targeted modes produce consistently more crashing executions than any of the baselines. The p-scores for the modified versus their baseline counterparts are low, which indicates a considerable difference. As expected, the difference is more apparent in the bottom two modes where the ad-hoc budget extension was disabled, i.e. when the fuzzing budget is fixed by a fitness function and not allowed to increase dynamically.

Although the modified modes did not specifically target uniqueness, they achieved more unique crashes nonetheless.

Mode	Total crashes	Unique crashes	Total paths
B1 / T1	2.250 / **4.269**	3.231 / **3.654**	2.981 / **3.385**
	< 0.0001	0.3897	0.4534
B2 / T2	3.212 / **4.462**	3.500 / **3.981**	2.539 / **3.692**
	0.0107	0.2472	0.0261
B3 / T3	2.654 / **4.539**	3.192 / **3.654**	3.808 / **4.481**
	< 0.0001	0.3727	0.1786
B4 / T4	2.058 / **4.558**	3.00 / **3.789**	3.442 / **3.673**
	< 0.0001	0.1032	0.6084

TABLE II: Mean rank score of each mode. The top row is the mean rank score by the baseline mode, and its modified counterpart in the range of $0-7$. The bottom row is the p-value for the two modes.

The modified modes found most unique crashes quicker than baselines in 63.5% of cases. More crashes appears to translate to discovering more distinct crashes, as indicated by a better performance in terms of unique crashes discovered. The significance of this difference is much smaller than for total crashes, as indicated by the higher p-values. Interestingly, the difference is more significant (lower p-scores) in the constant budget modes (B2/T2, B4/T4) than in those with built-in AFL prioritisation (B1/T1, B3/T3). In other words, the search methods are more effective at finding unique crashes than constant budget baselines but fare only somewhat better when used to augment existing AFL heuristics.

The modified modes perform somewhat better than baselines in terms of paths found. This is surprising as targeting crash discovery was expected to have diminished exploration performance. The differences however are not very significant, as suggested by the high p-values (except for B2/T2 comparison which may be an outlier). A possible explanation for this marginally improved performance is the modified strategies' penalising effect for non-suspicious executions, that also happen to be trivial and thus poorly suited to be fuzzing seeds. A more in-depth analysis of this effect is outside the scope of this work.

RQ3 *Generalisability of Representation*

The correlation between C library calls and crashes appears to generalise reasonably well to real world programs. The fitness landscapes based on a Codeflaws trained model however are not very convenient. This makes a targeted search strategy with the chosen representation unfit for testing arbitrary real world programs.

RQ3.1 *Correlation Generalises:* The results show that the correlation between the traces and crashes generalises well to previously unseen programs. The AUCs are 0.907, 0.880 and 0.612 for VLC, libjpeg and mpg321 respectively. The value of 0.612 for mpg321 indicates a weakness of the model, and ought to be improved – either by training on a more diverse corpus or by considering additional observations.

Inconvenient Fitness Landscape: An analysis of the values produced by the neural network revealed an issue with the representation in VLC and mpg321. The landscape had many plateaus: in the whole dataset of "interesting" inputs generated by AFL, there were only 183 and 92 unique crash

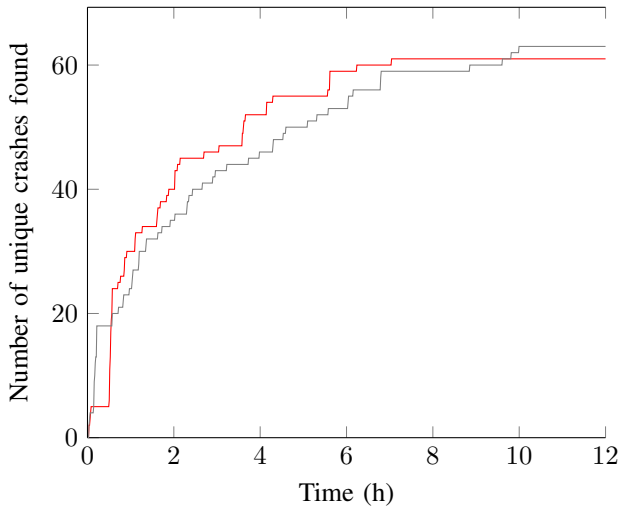

Fig. 8: Number of unique crashes discovered by the targeted mode **T4** (red) and the baseline mode mode **B4** (black). Interestingly, the pattern of discovery rates was also similar for paths and total crashes.

likelihood values for VLC and mpg321 respectively – *vs* 14853 for libjpeg. This may be due to the fact that these programs use standard C libraries to a lesser extent. A richer representation is thus necessary to analyse programs that do not use standard C libraries as extensively as those in Codeflaws. We aim to investigate both of these weaknesses in future work.

RQ3.2 *Targeted Search Consistent with RQ2:* A search across landscapes with plateaus would have been meaningless, so targeted search experiments were only conducted on libjpeg. In terms of unique crashes, total paths and total crashes, the targeted strategy found a median of 1267, 61 and 13681, versus 1313, 63 and 8542 found by a baseline. The number of paths found and unique crashes found are similar in both modes, but the targeted strategy reached far more total crashes. This is indicative of the effect of a targeted strategy on fuzzing given a sufficiently rich representation. These numbers are in line with those of the Codeflaws experiments.

Pattern of Crash and Path Discovery: The rate of discovery of crashes was consistent across all fuzzing runs, both for the targeted mode and the baseline mode. For the first ca. 10% of the fuzzing process the baseline outperforms the targeted strategy. In 10% to 50% of fuzzing, the targeted strategy gained considerably over the baseline. Finally, for the second half, the baseline picked up again. This behaviour can be attributed to the fact that while there are very few seeds (the process was started with a blank file), heavily prioritising one over others limits exploration. Once more paths are found, focusing on the most suspicious ones yields the most benefit. Finally, when discovery stalls, exploration becomes once again more viable. An example of this effect is shown in Fig. 8. These observations are not central findings of this study, but they are interesting for potential future work using alternative representations and search strategies.

VI. Related Work

SBST has been widely applied [21], [22], [23], [24], but it has limitations [25]. Fitness functions in SBST typically aim to improve coverage criteria [4], rather than targeting a specific property or behaviour of executions or programs [26]. The effectiveness of search for coverage criteria scores has been high although the effectiveness of coverage criteria in creating test sets good for fault discovery has recently been disputed [5], [6], and even shown to be detrimental as a sole target [7], [8], [9]. In this work, we instead explore the idea of using fitness functions that target execution properties in the context of SBST.

Search strategies that target execution properties are not unknown in SBST, although they are less common than ones that target coverage. Search methods have been applied to produce a test suite explicitly targeting defects, albeit the intermediate fitness was coverage of mutated code [27]. More recently, Rojas *et al.* have used exception coverage to formulate a fitness function specifically targeting exceptions [28]. Work on crash reproduction has considered the presence of a crash and the similarity of a crashing execution's stack trace to that of a candidate solution [29], [30]. While our approach also uses crashes and function calls as part of the fitness function, its construction is fundamentally different. Non-functional properties like execution time [31] and memory usage [32] have also been targeted. Search based approaches have also been used in exploit generation. For instance, a timing attack on the RSA encryption [33] deduces the correct password by observing the time it takes for a program to deny access to a guess [34] with the time channel acting as a fitness function. The use of time in the fitness function here relies on knowledge of the implementation of RSA. In our work, a fitness function targeting an execution property is constructed without knowledge of the underlying implementation.

A popular method in SBST is fuzzing [35]. Fuzzing is a technique which attempts to crash a program by feeding it randomly mutated inputs [36]. Modern fuzzers use program profiling and feedback to guide their strategy, and they have also been used to test for properties other than crashes [37]. One such fuzzer is the widely used American Fuzzy Lop (AFL) [10]. AFL has recently gathered much attention in academia: it has been improved using a number of techniques. Symbolic execution was used to solve cases where AFL's internal heuristics were ineffective [38]. Its coverage driven heuristics have been improved [39]. Application-specific magic value awareness has been introduced [40]. Context sensitivity, taint tracking and search based on gradient descent have also been explored [41]. Its mutation strategies have been augmented using a neural network that approximates a program's semantics w.r.t. input bytes [42]. AFL is also used as the underlying tool in this work. The methodology here is not aimed at immediate improvement to AFL, even if this is a consequence. It is instead an exploration of a conceptually novel approach to driving a fuzzing process with a machine learning based fitness function.

ML techniques have long been used in software engineering. Neural networks were employed for test suite optimisation as early as 1995 [43]. More recently, ML has been used for test suite refinement [44], fault localisation [45] and guided model based testing [46]. A generative neural network was applied to SBST for generating well-formed pdf files to test a pdf reader [47]. A generative model has also been used for producing seeds for AFL [48]. In that work, a model is trained per file format: for mp3, bmp and flv. These models are then used to produce new seeds for AFL. The use of ML for generative purposes in the context of search for execution properties is a planned extension to our work.

There are also examples of execution trace profiling and analysis with ML methods. Forrest *et al.* used statistical methods to identify abnormal behaviour in Linux processes [49]. That work is an example of anomaly detection whose purpose is to discover undesirable behaviour. A comprehensive survey of tools related to anomaly detection was produced by Chandola et al. [50]. Other relevant work includes behaviour classification using CFG traces [51], test input classification and generation [52], and bug detection [53]. Our work analyses execution traces of programs with ML tools. Unlike the work mentioned above however, we use this analysis to construct a fitness function for an SBST process.

VII. Conclusion and Future Work

In this paper we propose a novel approach for constructing a property targeting fitness function using ML. This is done by first training a classifier on observations of a corpus of crashing and non-crashing library call execution traces across a range of various C programs. The classifier provides crash likelihood estimates to a fuzzer which prioritises *non-crashing* candidate solutions based on this value. Results show that a fitness function constructed in this way has a clear effect on the rate of crash discovery.

The results were obtained with two sets of experiments. The first one shows that the execution traces represented by standard C library calls are strongly correlated with the presence of a crash. The area under curve of the ROC measure is 0.901. This is evidence of the representation being potentially useful for constructing a fitness function specifically targeting crashes. The second set of experiments shows how the crash likelihood estimate can be used as a fitness function. On a test set of programs, a crash targeting fuzzing strategy based on the crash likelihood outperformed a baseline in 80.1%, 63.5% and 61.5% of cases with respect to total crashes, unique crashes and total paths discovered respectively. The methodology backed by the experimental evidence presented in this work is expected to provide ample opportunity for further research.

The general structure of our approach is the following: collect execution traces, train an ML mechanism to predict a property from these traces, and then use the prediction as a component in a fitness function for an SBST tool. The approach is viable if there is a pattern in the representation that correlates with the property and the ML technique can learn to identify this pattern. Several aspects of the approach can be investigated further. First, other observations and representations can be investigated. In this paper we showed that while the C library call representation works for the set of test programs that are similar to the corpus, it has limited efficacy when applied to some programs. This shortcoming may be overcome by an alternative representation. Second, targeting properties other than crashes should be investigated in a similar way. This may include information leaks, exploits, specific crash types and different resource usages. Current work on alternative representations and properties has promising results. Third, rather than simply weighting a fuzzing budget with a crash likelihood score, a fitness function may be constructed differently. For instance, a fitness function need not be based on a single value like the crash likelihood given a trace of C library calls. Instead, it may be a combination of multiple targets based on a representation of several traces. The traces do not need to be specific to C or any other language but rather depend on the profiling tool used. Furthermore, the fitness does not need to be used in AFL – any other SBST input generator that uses a fitness function to evaluate intermediate results ought to be appropriate.

Acknowledgements

In the conduct of this research Joffe was supported by the EPSRC DAASE project (EP/J017515) and Clark was partially supported by the EPSRC InfoTestSS project (EP/P005888). The hardware for conducting the experiments was provided by Nvidia as part of their academic grant programme.

References

[1] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. Mcminn, A. Bertolino *et al.*, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[2] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[3] M. Harman and B. F. Jones, "Search-based software engineering," *Information and software Technology*, vol. 43, no. 14, pp. 833–839, 2001.

[4] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2016.

[5] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 435–445.

[6] P. S. Kochhar, F. Thung, and D. Lo, "Code coverage and test suite effectiveness: Empirical study with real bugs in large systems," in *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*. IEEE, 2015, pp. 560–564.

[7] G. Gay, M. Staats, M. Whalen, and M. P. Heimdahl, "The risks of coverage-directed test case generation," *IEEE Transactions on Software Engineering*, vol. 41, no. 8, pp. 803–819, 2015.

[8] M. Staats, G. Gay, M. Whalen, and M. Heimdahl, "On the danger of coverage directed test case generation," in *Fundamental Approaches to Software Engineering*. Springer, 2012, pp. 409–424.

[9] M. P. E. Heimdahl, D. George, and R. Weber, "Specification test coverage adequacy criteria= specification test generation inadequacy criteria," in *High Assurance Systems Engineering, 2004. Proceedings. Eighth IEEE International Symposium on*. IEEE, 2004, pp. 178–186.

[10] M. Zalewski, "American fuzzy lop," 2007, http://lcamtuf.coredump.cx/afl/.

[11] ——, "Binary fuzzing strategies: What works and what doesn't," 2014, https://lcamtuf.blogspot.co.uk/2014/08/binary-fuzzing-strategies-what-works.html.

[12] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *ACM Sigplan notices*, vol. 42, no. 6. ACM, 2007, pp. 89–100.

[13] V. P. . Users, 2017, http://valgrind.org/gallery/users.html.

[14] V. Developers, "Callgrind: a call-graph generating cache and branch prediction profiler," 2017, http://valgrind.org/docs/manual/cl-manual.html.

[15] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[16] R. M. Zur, Y. Jiang, L. L. Pesce, and K. Drukker, "Noise injection for training artificial neural networks: A comparison with weight decay and early stopping," *Medical physics*, vol. 36, no. 10, pp. 4810–4818, 2009.

[17] C. Cortes and M. Mohri, "Auc optimization vs. error rate minimization," in *Advances in neural information processing systems*, 2004, pp. 313–320.

[18] S. H. Tan and S. Mechtaev, "Defects in codeflaws," 2017, https://codeflaws.github.io/.

[19] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE transactions on software engineering*, vol. 41, no. 5, pp. 507–525, 2015.

[20] F. Chollet *et al.*, "Keras: The python deep learning library," 2018, https://keras.io/.

[21] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 416–419.

[22] ——, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2013.

[23] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A systematic review of the application and empirical investigation of search-based test case generation," *IEEE Transactions on Software Engineering*, vol. 36, no. 6, pp. 742–762, 2010.

[24] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *science*, vol. 220, no. 4598, pp. 671–680, 1983.

[25] P. McMinn, "Search-based software testing: Past, present and future," in *Software testing, verification and validation workshops (icstw), 2011 ieee fourth international conference on*. IEEE, 2011, pp. 153–163.

[26] A. Aleti, I. Moser, and L. Grunske, "Analysing the fitness landscape of search-based software testing problems," *Automated Software Engineering*, vol. 24, no. 3, pp. 603–621, 2017.

[27] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *IEEE Transactions on Software Engineering*, vol. 38, no. 2, pp. 278–292, 2012.

[28] J. M. Rojas, J. Campos, M. Vivanti, G. Fraser, and A. Arcuri, "Combining multiple coverage criteria in search-based unit test generation," in *International Symposium on Search Based Software Engineering*. Springer, 2015, pp. 93–108.

[29] F. M. Kifetew, W. Jin, R. Tiella, A. Orso, and P. Tonella, "Reproducing field failures for programs with complex grammar-based input," in *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*. IEEE, 2014, pp. 163–172.

[30] M. Soltani, A. Panichella, and A. van Deursen, "A guided genetic algorithm for automated crash reproduction," in *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 2017, pp. 209–220.

[31] J. Wegener and M. Grochtmann, "Verifying timing constraints of real-time systems by means of evolutionary testing," *Real-Time Systems*, vol. 15, no. 3, pp. 275–298, 1998.

[32] F. Wu, W. Weimer, M. Harman, Y. Jia, and J. Krinke, "Deep parameter optimisation," in *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. ACM, 2015, pp. 1375–1382.

[33] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[34] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Annual International Cryptology Conference*. Springer, 1996, pp. 104–113.

[35] J. Li, B. Zhao, and C. Zhang, "Fuzzing: a survey," *Cybersecurity*, vol. 1, no. 1, p. 6, 2018.

[36] M. Sutton, A. Greene, and P. Amini, *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[37] K. Claessen and J. Hughes, "Quickcheck: a lightweight tool for random testing of haskell programs," *Acm sigplan notices*, vol. 46, no. 4, pp. 53–64, 2011.

[38] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *NDSS*, vol. 16, 2016, pp. 1–16.

[39] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, 2017.

[40] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[41] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," *arXiv preprint arXiv:1803.01307*, 2018.

[42] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "Neuzz: Efficient fuzzing with neural program learning," *arXiv preprint arXiv:1807.05620*, 2018.

[43] C. Anderson, A. Von Mayrhauser, and R. Mraz, "On the use of neural networks to guide software testing activities," in *Test Conference, 1995. Proceedings., International*. IEEE, 1995, pp. 720–729.

[44] L. C. Briand, Y. Labiche, and Z. Bawar, "Using machine learning to refine black-box test specifications and test suites," in *Quality Software, 2008. QSIC'08. The Eighth International Conference on*. IEEE, 2008, pp. 135–144.

[45] L. C. Briand, Y. Labiche, and X. Liu, "Using machine learning to support debugging with tarantula," in *Software Reliability, 2007. ISSRE'07. The 18th IEEE International Symposium on*. IEEE, 2007, pp. 137–146.

[46] R. Ben Abdessalem, S. Nejati, L. C. Briand, and T. Stifter, "Testing advanced driver assistance systems using multi-objective search and neural networks," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 63–74.

[47] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 50–59.

[48] C. Lv, S. Ji, Y. Li, J. Zhou, J. Chen, P. Zhou, and J. Chen, "Smartseed: Smart seed generation for efficient fuzzing," *arXiv preprint arXiv:1807.02606*, 2018.

[49] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for unix processes," in *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*. IEEE, 1996, pp. 120–128.

[50] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM computing surveys (CSUR)*, vol. 41, no. 3, p. 15, 2009.

[51] J. F. Bowring, J. M. Rehg, and M. J. Harrold, "Active learning for automatic classification of software behavior," in *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 4. ACM, 2004, pp. 195–205.

[52] C. Pacheco and M. D. Ernst, "Eclat: Automatic generation and classification of test inputs," in *European Conference on Object-Oriented Programming*. Springer, 2005, pp. 504–527.

[53] D. Lo, H. Cheng, J. Han, S.-C. Khoo, and C. Sun, "Classification of software behaviors for failure detection: a discriminative pattern mining approach," in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2009, pp. 557–566.