Reducing Test Cases with Attention Mechanism of Neural Networks

Xing Zhang, Jiongyi Chen*, Chao Feng, Ruilin Li, Yunfei Su, Bin Zhang, Jing Lei, and Chaojing Tang

National University of Defense Technology

Abstract

As fuzzing techniques become more effective at triggering program crashes, how to triage crashes with less human efforts has become increasingly imperative. To this aim, test case reduction which reduces a crashing input to its minimal form plays an important role, especially when analyzing programs with random, complex, or large inputs. However, existing solutions rely on random algorithms or pre-defined rules, which are inaccurate and error-prone in many cases because of the implementation variance in program internals.

In this paper, we present SCREAM, a new approach that leverages neural networks to reduce test cases. In particular, by feeding the network with a program's crashing inputs and non-crashing inputs, the network learns to approximate the computation from the program entry point to the crash point and implicitly denotes the input bytes that are significant to the crash. With the invisibility of the trained network's parameters, we leverage the attention mechanism to explain the network, namely extracting the significance of each input byte to the crash. At the end, the significant input bytes are re-assembled as the failure-inducing input.

The cost of our approach is to design a proper dataset augmentation algorithm and a suitable network structure. To this end, we develop a unique dataset augmentation technique that can generate adequate and highly-differentiable samples and expand the search space of crashing input. Highlights of our research also include a novel network structure that can capture dependence of input blocks in long sequences.

We evaluated SCREAM on 41 representative programs. The results show that SCREAM outperforms state-of-the-art solutions regarding accuracy and efficiency. Such improvement is made possible by the network's capability to summarize the significance of input bytes from multiple rounds of mutation, which tolerates perturbation occurred in random reduction of single crashing input.

*Corresponding author

1 Introduction

To discover and eliminate software vulnerabilities, fuzzing nowadays has been considered one of the most effective approaches by randomly or strategically generating a large number of inputs to feed the program, exploring program paths as many as possible, and hopefully triggering program exceptions. For the past decade, there has been a series of research on fuzzing (e.g., [13, 14, 23, 24, 31, 43]), demonstrating significant effectiveness in triggering crashes. However, with more crashing inputs produced by fuzzers, the challenge comes as analysts often need to spend plenty of time to trace the crashing inputs step-by-step and inspect the program logic, in order to understand the root cause of the discovered crashes [22, 39, 51]. Even worse, fuzzers tend to generate inputs in their most ill-formed and peculiar shape, in an attempt to cover corner paths and trigger unexpected crashes. Such inputs often add heavy burden on subsequent debugging procedures, by misleading analysts to dive into unnecessary program logic that is not related to the crash. In fact, only a small portion of the crashing inputs are necessary to reproduce the failure.

Challenges in test case reduction. Test case reduction [25, 33, 36] which aims to minimize crashing inputs by removing irrelevant portion and preserving failure-inducing portion, plays an important role in facilitating debugging tasks like crash analyses [9, 12, 19, 38]. Prior efforts to reduce test cases, on one hand, rely on random reduction. A prominent example is delta debugging [8, 50], which adopts various search strategies (e.g., binary search) to randomly reduce inputs by gradually increasing the granularity of reduction and confirming whether the crash can be reproduced. However, such an approach only achieves local minimum of reduction and cannot reduce discontinuous input blocks that correlate with each other [2]. For instance, when there are interdependent input blocks that typically appear in file-based crashing inputs, those related blocks should either be preserved or reduced at the same time. This is a difficult task for random reduction-based approaches as they lack deep understanding of pro-

gram logic. On the other hand, rule-based approaches, such as information flow tracking [16, 21, 30] and input structure-aware reduction [33], require analysts to manually specify rules about program semantics. For instance, information flow tracking-based approaches (e.g., taint analysis) attempt to recover accurate information flows between inputs bytes and the crash, to precisely determine a subset of input that actually affects the crash. This process could be inaccurate and error-prone, because the recovery of information flow is built upon the comprehensive understanding of a crash, which involves expert knowledge.

Our approach. At the core, test case reduction is to determine a subset of input that actually contributes to a crash. In our research, we treat test case reduction as a deep learning task and leverage neural networks to denote "essential" input bytes as significant to a crash and denote "accidental" input bytes as insignificant. Particularly, by feeding the network with crashing inputs and non-crashing inputs, the network learns to approximate the computation from the program entry point to the crash point. However, as the trained network's parameters are not understandable, such valuable description of contribution is hidden in the network. To explain the trained network, we utilize the interpretability of neural networks (by adopting the attention mechanism) to extract the input weights that denote the contribution to the crash. In the end, we reassemble significant input bytes as the failure-inducing input according to the calculated contribution.

A proper dataset and a suitable network structure are vital to the success of deep learning tasks. This is also the cost of our approach that does not rely on digging into program internals. Our research conquers several challenges in the adoption of deep learning-based reduction. On one hand, since there is no dataset that can be directly used for our purpose, we develop an online dataset augmentation algorithm to mutate a single crashing input and output adequate positive and negative samples during the training process. This algorithm works in conjunction with the neural network and helps the network to gradually achieve a fitting state: when each round of network training is completed, the significance of input byte calculated by the trained network is used to confine the focus of mutation on crashing inputs in the next round of sample generation. In this way, it is more likely to produce a different crashing input as the seed to breed variant samples and thus expand the sample space of crashing input. Consequently, the neural network can draw a more accurate boundary between crashing inputs and non-crashing inputs. On the other hand, existing network architectures only deal with short sequences such as sentences of natural language. Handling program inputs with tens of thousands of bytes presents a new challenge. As such, we design a new network architecture that is able to capture and preserve interdependence of input blocks in long input sequences, by combining convolutional layers and recurrent layers in an innovative way.

We implemented the prototype of SCREAM (teSt Case REduction with Attention Mechanism) and evaluated it on 41 representative programs including 29 CGC programs and 12 real-world programs. The evaluation demonstrates that SCREAM is highly effective and accurate in test case reduction. It achieves an average reduction rate of 75.4% which takes 29.8 minutes on average. More importantly, SCREAM has no false positives and less false negatives when compared with the state-of-the-art solutions—afl-tmin [1], Picireny [4], and Penumbra [21]. With the help of SCREAM, 70.7% of the reduced inputs have reached ground truth. This is attributed to SCREAM's capability to solve control flow complexity to some extent by continuously mutating a subset of inputs guided by the calculated significance. Furthermore, compared with AFL's mutation engine, our dataset augmentation algorithm facilitates SCREAM to achieve higher reduction efficiency even when SCREAM is fed with less samples. The amount of samples generated by SCREAM's algorithm is only 38.0% of that generated by AFL's algorithm. Regarding interpretability methods, the attention mechanism that we adopted outperforms partial derivatives in reduction efficiency.

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

- **New insights.** We leverage the neural network to address the problem of test case reduction. Our intuition is to train the network to approximate the computation from the program entry point to the crash point and leverage the interpretability to denote failure-inducing input bytes that are significant to the crash.
- **New techniques.** We present several new techniques to address the challenges in designing the neural network-based solution. In particular, we design a new dataset augmentation algorithm that works in conjunction with the neural network and generates adequate and high-differentiable samples to expand the space of crashing input. Besides, we also present a new architecture of neural network that can process sequence information for long inputs.
- **Evaluation.** We evaluated SCREAM[1] on 41 programs, including 29 CGC programs and 12 real-world programs. The overall results show that SCREAM is more efficient and accurate than the state-of-the-art solutions.

2 Attention Mechanism for Interpretability

The attention mechanism was originally proposed to improve the fitting of neural networks by assigning different weights to the input sequence and minimizing the loss function [17]. Recent years a line of research [10, 27, 32, 48] leveraged the attention mechanism for the interpretability of neural networks, allowing us to directly inspect the internal working of neural networks. The hypothesis is that the magnitude of attention

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

weights highly correlates with how relevant a specific region of input is, for the prediction of output at each position in a sequence. This can be easily accomplished by visualizing the attention weights for a set of input and output pairs. In this paper, we borrow this idea and leverage the attention mechanism to visualize the contribution of each input region to the output.

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

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

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

$$\begin{aligned} \vec{\alpha} = g(\vec{x};\vec{\theta}), \vec{v} = \alpha_1\vec{x}_1 + \alpha_2\vec{x}_2 + ... + \alpha_n\vec{x}_n, \\ y = f(\vec{v};\vec{\theta}), where \sum_i \alpha_i = 1, \alpha_i > 0 \end{aligned} \tag{2}$$

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

$$L(f(\vec{x},\vec{\theta})) = \sum_i |f(g(\vec{x}^i;\vec{\theta}) \odot \vec{x}^i;\vec{\theta}) - y^i|^2, s.t. \sum g(\vec{x}^i;\theta) = 1 \tag{3}$$

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

$$\begin{aligned} \vec{\alpha} = softmax(g(\vec{x};\vec{\theta})), \\ \vec{v} = \alpha_1\vec{x}_1 + \alpha_2\vec{x}_2 + ... + \alpha_n\vec{x}_n, y = f(\vec{v};\vec{\theta}), \\ softmax(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \end{aligned} \tag{4}$$

And the loss function becomes:

$$L(\vec{\theta}) = \sum_i |f(softmax(g(\vec{x}^i;\vec{\theta})) \odot \vec{x}^i;\vec{\theta})) - y^i|^2 \tag{5}$$

The network that we designed (as described in Section 4.3) follows the above transition equation. In fact, Equation (4) is the core architecture of the attention mechanism and such an architecture can be used to determine the relevance of the input bytes and the output. In particular, under this architecture, we are able to get the $\vec{\theta}$ by minimizing $L(f(\vec{x},\vec{\theta}))$ with the gradient descent. Function $g(\vec{x};\vec{\theta})$ or $f(\vec{x};\vec{\theta})$ could be convolutional neural networks (CNN), recurrent neural networks (RNN) or fully connected networks. While in seq2seq networks, $g(\vec{x};\vec{\theta})$ is LSTM and $f(\vec{x};\vec{\theta})$ is a fully connected network.

3 Test Case Reduction

Given a program P and a crashing input $\vec{x} = (x_1, x_2, .., x_n)$ that causes crash C, the goal of test case reduction is to find a minimal subset of the crashing input $\vec{x} = (x_i, .., x_j), (1 \leq i \leq j \leq n)$ that triggers the same crash C of program P. The output of test case reduction is also known as the failure-inducing input.

3.1 Motivating Example

To better understand the problem of test case reduction, we use an example to illustrate and compare existing solutions and our solution. Listing 1 shows a code snippet that we captured and simplified from a real-world program that digests file-based inputs. Assume that the fuzzer produces the following test case: "`I1S[AAAAI2A]y2SS1SI3`" (in this example, the minimal form of crashing input is "`I1I2I3`"). As the code does not check the size of *idArray*, this input causes out-of-bounds access on the third integer assignment to *idArray* at line 22 (the array can at most hold two integers).

```
1  char* input = scanf();
2  int ptr, iPtr = 0;
3  char* Str_Storage = "";
4  int idArray[2];
5  while(true){
6    if(input[ptr++] == 'S'){
7        if(input[ptr++] == '['){
8          while(true){
9            if(input[ptr++] == 'I')
10             if((byte)input[ptr] < 10)
11               idArray[iPtr++]=(byte)input[ptr];
12             ptr++;
13           else if(input[ptr++]==']')
14             break;
15           else{Str_Storage += input[ptr++];}
16         }
17       }else{
18         Str_Storage += input[ptr+3];
19         ptr = ptr+3;}
20   }else if(input[ptr++] == 'I'){
21       if((byte)input[ptr] < 10)
22         idArray[iPtr++] = (byte)input[ptr];
23       ptr++;
24   }else if(input[ptr++]=='['|input[ptr++]==']')
25         Syntax_Error_Exit();
26 }
```

Listing 1: Example code

Existing techniques. Random reduction does not analyze program internals logic at all. Delta debugging, for example, adopts binary search and increases granularity to determine

Table 1: Example Dataset

Positive Samples	Negative Samples
I1S[AAAAI2A]y2SS14I9	*I1S[AAAAIIWD2SS1SIW*
I0S[ZD$EI2!]I2SCEDI5	*IWQeAAAAI2A]EDSS1SI3*
I7S[WS∗dI23]I5Sy3uI8	*IFS8yS∗dI23]I5SyPoie*
I5S[WSidI23]I0Wy3uI6	*iueS[WSidwejI0Wy3uIN*
I6DDeikDPfeI5OM82LI7	*IXDDeipqioI5OM82efs*
I4TypwqCv34I2OEpvBI8	*eoibpwqCv34feOEpvBdw*
...	...
I9lqdvbmn13I1hzxw8I7	*we4qdvbmn13zhhzxw8ep*

I	1	S	[	A	A	A	A	I	2	A	]	y	2	S	S	1	S	I	3
.25	.05	.021	.02	.003	.003	.003	.003	.25	.05	.003	.02	.003	.003	.006	.005	.003	.005	.25	.05

Figure 1: Example weights of trained network

failure-inducing input sets. However, a fundamental drawback is that it does not consider interdependence among discontinuous input blocks and therefore only achieves local minimal reduction. When discontinuous input blocks correlate with each other, they should be considered as a whole in test case reduction. In the example, reducing "`S[`" or "`]`" separately does not lead to the crash. The code checks the paired keywords "`[`" and "`]`", indicating that "`S[`" and "`]`" must be reduced together.

On the other hand, for rule-based approaches, the program execution from the entry point to the crash is described by a set of logical expressions. The program's execution is analyzed with operational semantics at the instruction-level. For instance, one can use backward dynamic taint analysis to determine the information flow between the input bytes and the crash, by marking crash points as taint sources and marking input bytes as taint sinks. However, a drawback of this approach is that it is often difficult to precisely define taint sources and taint policy when analyzing the root cause [21], which leads to imprecision in information flow tracking. For example, as can be seen in Listing 1, when defining the out-of-bounds byte of *idArray* as the taint source and defining the input as the taint sink, conservative taint policy would cause undertainting and produce "`I3`" as the result. Nevertheless, when trying to include "`I1I2`" in the output, non-conservative taint policy would lead to overtainting and produce "`S[]`" as a side effect. Reducing inputs in both ways does not trigger any crashes. Therefore, taint-analysis-based approaches are less effective in test case reduction.

3.2 Our Insight

In this paper, we aim to address the challenge from a new angle: conceptually, test case reduction is to determine a subset of the input that contributes to the crash. Therefore, we utilize the neural network to approximate the computation from the program entry point to the crash point. Approximating such computation is a numerical optimization problem (involves arithmetic expressions rather than logical expressions) that is achieved by minimizing errors described by mathematical loss functions. Instead of directly determining the "essential" bytes in the crashing input, the purpose of fitting/approximation is to let the network differentiate crashing inputs with non-crashing inputs and activate the input nodes that contribute more to the crash. When the network is trained, we utilize the attention mechanism to extract the "essential" input bytes through its explanation on the trained network's internal.

Table 2: Reduction Process (with Binary Search)

Input Bytes with Weight	Crash?
`I1S[I2]I3` (0.25|0.05|0.021|0.02|0.25|0.05|0.02|0.25|0.05)	✓
`I1I2I3` (0.25|0.05|0.25|0.05|0.25|0.05)	✓
`III` (0.25|0.25|0.25)	×

More specifically, our neural network is trained in a supervised manner. The input of the network is the program input, and the output is the labeled data about whether the crash has been triggered. To feed the network, we design a novel dataset augmentation algorithm which works in conjunction with the network training and helps the network to achieve a fitting state: by mutating a single crashing input, the algorithm outputs a large set of samples in each round of mutation and training (the left column in Table 1 lists positive samples that can trigger the crash. The right column shows negative samples that do not trigger any crashes). After each round of network training, we leverage the attention mechanism to calculate the importance to the output for each input byte. The calculated significance score is then used to guide the mutation in the next round. During the training and mutation process, the weights of input bytes that contribute more to the crash increase, while the weights of less-contributed bytes are lowered. After a period, the weights become stable (Figure 1 shows an example weights of the trained network). In the end, we determine the reduced input by re-assembling the bytes according to their final weights. Table 2 illustrates the idea of how the reduced input is re-assembled.

The neural network plays an irreplaceable role in reduction: on one hand, it offers a way of guiding the mutation through weight adjustment. Without such guidance, the search space of reduction will largely expand and it is less likely to find a reduction strategy that captures the dependence among input blocks in the long input; On the other hand, the neural network accumulates knowledge about "crash contribution" by adjusting and summarizing the significance of input bytes from multiples rounds of training and mutation, which eventually instructs the one-shot reduction when the network is trained (details are described in Section 4.5).

Technical challenges. Nevertheless, our insights come with several challenges that should be addressed:

- The first problem is how to generate datasets that are suit-

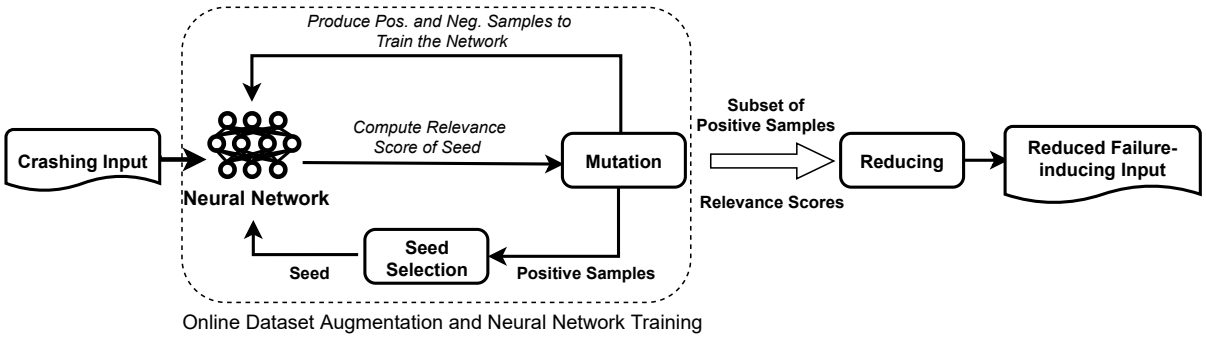

Figure 2: Overview of SCREAM

able for training. Fuzzers often produce a small amount of crashing inputs. The amount is inadequate for the network training. Therefore, we need to generate training samples by mutating existing crashing inputs. However, existing neural-network-based mutation for dataset augmentation is mainly designed for images. The mutation includes rotation, flipping, etc., which is not suitable for program inputs. The other line of mutation is for fuzzing (e.g., AFL's mutation), with the aim of discovering one path that extends current code coverage. In the consecutive rounds, such mutation changes a small portion of input that is "interesting", and the mutated input may potentially explore one more path. As a result, this kind of mutation produces test cases that are close to each other in the input space. Those clustered samples are not suitable for training the network, as it would cause overfitting. Therefore, to achieve a satisfying fitting state for the network, it is desirable to construct/augment the dataset with highly-differentiable samples that can expand the sample space.

- The second challenge is that existing RNN architectures that can process sequence information do not directly suit our task. On one hand, RNNs are mainly applied to natural language processing, which deals with short input sequences like sentences of natural language (the length is usually less than 150). However, for test case reduction, the input length often ranges from hundreds of bytes to tens of thousands of bytes. The long input tends to introduce vanishing gradient problems for RNNs and thus lead to underfitting. On the other hand, applying the attention mechanism would eliminate sequence information of input due to the sum operation in Eq (4) in Section 2, which would eventually affect the calculation of the input weight $\vec{\alpha}$. Therefore, we need to design a new RNN-based architecture that can capture dependence of input blocks in long input sequences and make the attention mechanism applicable.

Solutions. To address the above problems, we propose two novel techniques in this paper:

- We design an online dataset augmentation technique that can automatically construct the dataset with a single crashing input. The dataset is generated by mutating a given crashing input, and the generation process works collaboratively with the network training process. Initially the network takes a single crashing input and produces the relevance score that can guide the mutation. To select a seed for the next round of mutation, we use the bi-gram model to measure the similarity between input vectors and select the most dissimilar one from the current corpus as the seed. In doing so, the algorithm can generate highly-differentiable samples in input space to train the neural network.
- We design a new RNN-based network architecture to handle dependence of input blocks in long sequences. First, we utilize CNN to encode the one-dimensional long sequence to multi-dimensional feature vectors. Then we send the feature vectors to the RNN with the attention mechanism. As the input vector is compressed by CNN, back-propagation would make weight assignment less accurate. Therefore, we train multiple networks with different parameters at a time to reduce deviation and errors.

4 System Design

Figure 2 presents a high-level overview of our system. The core of SCREAM is a feedback system that plays the role of dataset augmentation and neural network training. First, the neural network is fed with a set of crashing inputs that are produced by the mutation component. The network outputs a relevance score that indicates the importance of input bytes. Then the relevance score is used to guide sample generation in the next round of mutation. As the mutation process is based on genetic algorithms, a seed is sent to the neural network to produce such a relevance score. In the meantime, the mutation component generates samples to enrich the dataset for training neural networks. Note that this iterative process will not affect the generalization of the network, since the generalization mainly depends on the quality of dataset, which is guaranteed

by our dataset augmentation algorithm. When the network is well trained, the one-shot reduction is applied to the input according to the computed relevance score. We determine that the network is well-trained or stable when one of the following two requirements are met:

- The failure-inducing inputs keep unchanged in several consecutive rounds of iteration, indicating that the reduction does not make new progress at this stage.
- Referring to Section 4.2, the mutation algorithm generates either all positive samples or all negative samples. It means that the network becomes stable and is able to differentiate significant input fields and insignificant fields for the fed inputs.

4.1 Input and Output Embedding

One-hot vector is a popular approach of input embedding that has been widely used in many applications [3, 15]. However, it does not suit our case, as the input for the program to digest is often large, causing much computational inefficiency. On the contrary, we use real-valued vectors to encode the input so that it can be easily accepted by the network. In other words, any types of inputs (e.g., file-based inputs, string-based inputs) are directly converted into hexadecimal byte sequences.

The output of neural network is a boolean variable that represents whether a crash is triggered by the program input (i.e., labeling samples by executing the program with the given program inputs). The crashing input is marked with a positive label in the output. Moreover, we uniquely represent a crash using a short sequence of executed function calls starting backward from the crash point of the program. Note that such crash representation for data labeling does not need to be 100% accurate. Thanks to the tolerance of neural network-based approaches, as long as most of the samples are correct labeled in the training dataset, the neural network can still achieve a good fitting state.

4.2 Dataset Augmentation

A comprehensive dataset is critical to the training of neural networks. Given that positive samples are usually inadequate for a training dataset, we mutate crashing inputs rather than random inputs. In this way, there is a higher chance to produce positive samples. On the other hand, as illustrated in Figure 3, feeding the network with positive samples that are largely different from each other is desirable to the fitting of the network. However, existing fitness functions of mutation algorithms are mostly designed for exploring new program paths [41]. Those algorithms tend to choose new seed inputs that will potentially explore new paths in the next round. However, this would only mutate the "interesting" fields and cover a small portion of the input. As a consequence, the mutation algorithms would produce program inputs with high similarity in most input fields, leading to overfitting for the trained network.

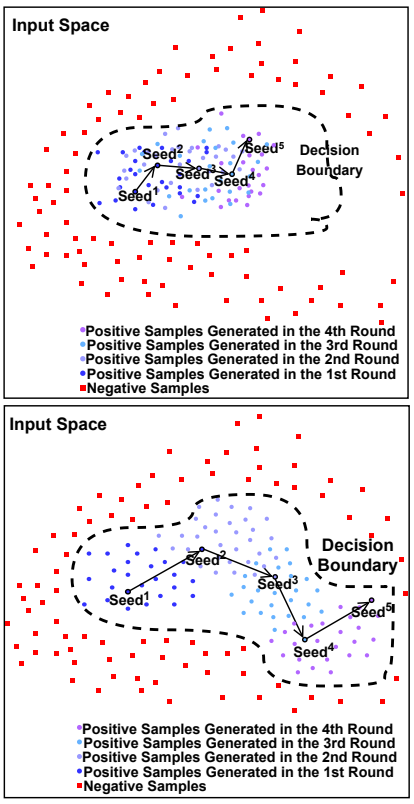

Figure 3: The distribution of the dataset generated by: existing approaches (on the top) and our approach (on the bottom). The samples generated by our approach are more scattered. Thus, the trained network is less likely to overfit

To tackle the challenge described above, we design a novel algorithm that generates training datasets with highly-differentiable samples. On the whole, the algorithm (as shown in Algorithm 1) is based on genetic algorithms and works in conjunction with the neural network. It consists of two parts: (1) mutation (i.e., which input fields to mutate and how to mutate those fields); (2) seed selection (i.e., how to select the seed for the next round). Below we give a detailed description of the algorithm.

Mutation. For each round of mutation, initially the neural network takes a crashing input $\vec{x} = (x_1, x_2, ..., x_n)$ and outputs a vector of relevance scores —$\vec{r} = (r_1, r_2, ..., r_n)$—that marks the importance of each input byte to the crash. Then, we select a set of input bytes whose relevance scores fall into the middle range [2]. Random mutation (setting byte value from 0x00 to 0xFF) is applied on those bytes, generating a set of mutated inputs $s = \{\vec{x}_a, \vec{x}_b, ...\}$ that include positive candidates (i.e.,

[2] As we tested, with the relevance scores falling into a middle range (e.g., from [15%, 90%] to [30%, 50%]), the results are close and satisfying. Besides, determining an optimal range for one case does not give optimal result on another case. Here we empirically choose [20%, 60%], as shown in Algorithm 1.

Algorithm 1 Algorithm to generate training datasets

Require: $seed \leftarrow crashing\ input$
$R\_score \leftarrow$ relevance score of crashing input
1: $minThd \leftarrow Sort(R\_score)[len(seed) * 20\%]$
2: $maxThd \leftarrow Sort(R\_score)[len(seed) * 60\%]$
3: **for** i $\in range(len(input))$ **do**
4: **if** $R\_score[i] \in [minThd, maxThd]$ **then**
5: $mutate\_indices.append(i)$
6: **end if**
7: **end for**
8: $Random\_Mutate\_Base\_On\_List(mutate\_indices)$
9: $Execute\_And\_Label\_Inputs()$
10: $Gen\_Corpus(Positive\_Candidates)$
11: $BigramScores \leftarrow Get\_BiGram\_Scores(Positive\_Candidates)$
12: $NextRoundSeed \leftarrow Corpus[min(BigramScores)]$

inputs that trigger the crash) and negative candidates (i.e., inputs that do not trigger the crash).

Note that we choose to mutate the input bytes with middle relevance scores. This keeps important input fields unchanged and corrects errors, which assists the network to achieve a fitting state and helps produce meaningful samples:

- Important fields that have high relevance scores remain unchanged to some extent, as those fields are supposed to be kept; insignificant fields that have low scores, on the other hand, will be reduced in subsequent steps (described in Section 4.5).
- If the network assigns a high score to an insignificant field, the score will be lowered by the network in the next round (because the network has determined output that can give feedback), leading to mutation on the field. Similarly, if the network assigns a low score to an important field, the score will rise in the next round. Therefore, this important field will remain unchanged in the next round (when its relevance score exceeds *maxThd* in Algorithm 1).

Seed selection. In this step, the goal is to select a seed for the next round. The seed is supposed to be the most different one among positive candidates. However, existing approaches of similarity measurement, such as cosine similarity, cannot be directly applied to byte sequences because our encoding is simply a representation that contains no semantics to facilitate similarity measurement. To this end, we borrow the idea from NLP and use the Bi-gram model to measure the difference between input byte sequences.

The Bi-gram model is based on Markov theory and can be utilized to convert a byte sequence into a numeric value using the occurrence probability. Given a program input $\vec{x} = (x_1, x_2, ..., x_n)$, the occurrence of $\vec{x}$ is denoted as $p(\vec{x}) = p(x_1, x_2, .., x_n) = p(x_1)p(x_2|x_1)... \ p(x_n|x_{n-1}, x_{n-2}, x_1)$. To identify the most different one in the corpus, we only need to determine the sample that has the lowest $p(\vec{x})$. Given that the occurrence of x_i is only related to its preceding byte x_{i-1} in the Bi-gram model, the occurrence of $\vec{x}$ is $p(\vec{x}) \approx p(x_1)p(x_2|x_1)...p(x_n|x_{n-1})$. Based on the Bayes rule, the posterior probability $p(x_i|x_{i-1})$ equals to $\frac{p(x_i, x_{i-1})}{p(x_{i-1})}$, where $p(x_i|x_{i-1})$ can be calculated by $\frac{C(x_i, x_{i-1})}{C(x_{i-1})}$ and $C(x)$ function is the count of x in the corpus. Since the product would make $p(\vec{x})$ extremely small, we use log function to calculate $p(\vec{x})$, which is:

$$log(p(\vec{x})) = \underbrace{log(\frac{C(x_1)}{\sum_{i=1}^{n} C(x_i)})}_{log(p(x_1))} + \underbrace{log(\frac{C(x_2, x_1)}{C(x_1)})}_{log(p(x_2|x_1))} + ... + \underbrace{log(\frac{C(x_n, x_{n-1})}{C(x_{n-1})})}_{log(p(x_n|x_{n-1}))} \tag{6}$$

The result $log(p(\vec{x}))$ is the Bi-gram score of $\vec{x}$, which is used to measure the difference among positive candidates. In the end, we use the up-sampling [5] to balance the negative samples and positive samples.

4.3 Network Structure

As described in Section 3.2, the network should be able to process long input without the loss of sequence information. The architecture of our network is shown in Figure 4. In particular, on one hand, we adopt the convolutional network before the *LSTM* network to encode the one-dimensional long input into high-dimensional short vectors, for the purpose of processing long inputs. Apart from that, the vector that is sent to the $softmax$ function should preserve the sequence information of the input that is compressed by convolutional layers. For this purpose, we utilize the *LSTM* network as the similarity function.

In the beginning, the input is passed through multiple one-dimensional convolutional layers (*Conv*1*D*). The *Conv*1*D* works as an encoder, with each *Conv*1*D* layer encoding the adjacent elements of layer input into a high-dimensional vector. The jth output of ith layer $\vec{o}_j^i$ is denoted as follows:

$$\vec{o}_j^i = f_{Conv1D}^i(\vec{x}^i_{(stride^i-1)*j}, \vec{x}^i_{(stride^i-1)*j+1}, \cdots, \vec{x}^i_{(stride^i-1)*j+kernel^i}) \tag{7}$$

where $stride^i$ is the stride parameter of the ith layer, $kernel^i$ is the kernel parameter, $\vec{x}^i$ is the output of the $(i-1)$th layer, and $\vec{x}^0$ is the input of the network. Besides, the length of ith layer's output is denoted as $n^i = \lceil \frac{n^{i-1} - kernel^i + 1}{stride^i} \rceil$.

Assume that the input is $\vec{x} = (x_1, x_2, ..., x_n)$ where $\vec{x} \in \mathbb{R}^{n \times 1}$. After passing through m *Conv*1*D* layers, the output vector becomes $\vec{o}^m = (\vec{o}_1^m, \vec{o}_2^m, ..., \vec{o}_{n^m}^m)$, where $\vec{o}^m \in \mathbb{R}^{n^m \times filter^m}$ and $filter^m$ is the filter parameter of the mth layer. In this case, $\vec{o}^m$ can be regarded as the encoded vector of $\vec{x}$ with higher dimension and shorter length.

Then, the $\vec{o}^m$ is passed through the *LSTM* layer, and the jth output $\vec{o}_j^{LSTM}$ is $o_j^{LSTM} = f_{LSTM}(\vec{o}_j^m, o_{j-1}^{LSTM})$. This indicates

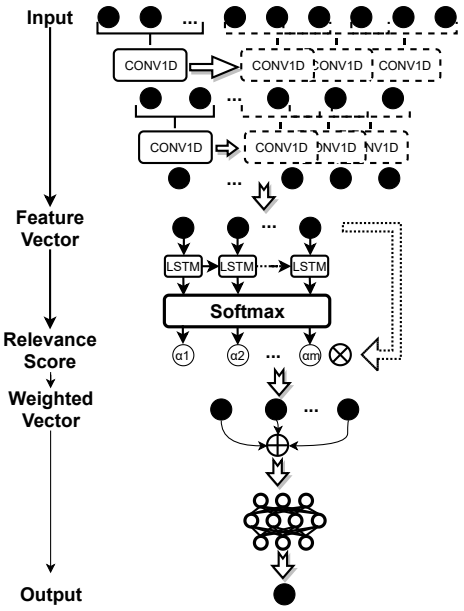

Figure 4: The network architecture

that $\vec{o}_j^{LSTM}$ vector contains sequence information of the current input node and all its previous input nodes. Therefore, the output vector $\vec{o}^{LSTM} = (o_1^{LSTM}, o_2^{LSTM}, ..., o_{n^m}^{LSTM})$ (where $\vec{o}^{LSTM} \in \mathbb{R}^{n \times 1}$) also preserves such sequence information.

In the end, after passing through the $softmax$ function, the vector of relevance score becomes $\vec{\alpha} = (\alpha_1, \alpha_2, .., \alpha_{n^m})$. For each input byte, we multiply the feature vector $\vec{o}^m$ by the vector of relevance score $\vec{\alpha}$. After that, we add all the products together and get the eigenvector $\vec{v} = \sum_{i=0}^{n^m} \vec{o}_i^m * \alpha_i$. The $\vec{v}$ is then passed through fully-connected layers.

4.4 Relevance Computation

As the convolutional network compresses the long input sequence, the generated relevance score vector $\vec{\alpha}$ is much shorter than the original long input. As a consequence, it is imprecise to represent the significance of input byte with $\vec{\alpha}$. Thus, we design weight backward allocation and accumulation of multiple networks to reduce imprecision.

Backward weight allocation. As indicated in Equation (7), the $Conv1D$ layer's output $\vec{o}_j^i$ is determined by several nodes of $\vec{x}^{i-1}$. As an example, in Figure 5, the output of the 5th layer $\vec{o}_1$ is determined by the input $(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$. For a typical $Conv1D$ network, the $kernel$ parameter is usually larger than $stride$ parameter, meaning that an input node contributes to multiple output nodes. In other words, the input nodes that contribute to a certain output node have variant weights. Based on this fact, we design the backward weight allocation algorithm shown in Algorithm 2. More specifically, we assume that initially every input node has the same contribution to each of its affected output nodes. Then the weights

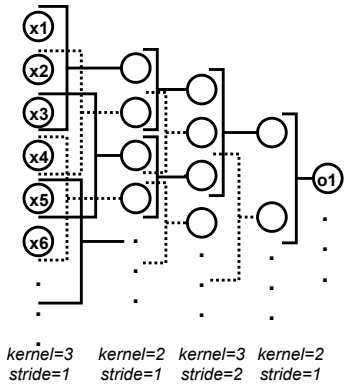

Figure 5: Illustration of $Conv1D$'s input and output. $\vec{o}1$ is influenced by $(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$ by Equation (7)

are recursively propagated from the last layer to previous layers. For instance, as shown in Figure 5, if the weight of $\vec{o}_1$ is set to 1.0, after backward propagation, the weights of input nodes are: $W_1^{\vec{o}_1} = 0.028, W_2^{\vec{o}_1} = 0.083, W_3^{\vec{o}_1} = 0.167, W_4^{\vec{o}_1} = 0.222, W_5^{\vec{o}_1} = 0.222, W_6^{\vec{o}_1} = 0.167, W_7^{\vec{o}_1} = 0.083, W_8^{\vec{o}_1} = 0.028$, where $W_i^{\vec{o}_j}$ is the weight of input node x_i assigned by output node $\vec{o}_j$.

Algorithm 2 Backward weight allocation algorithm

Require: $indices \leftarrow$ Input nodes share the same weight
$m \leftarrow$ The layer of $indices$
$\alpha^m \leftarrow$ The allocated weight on the mth layer of $indices$
$K \leftarrow$ Kernel parameter for each layer
$S \leftarrow$ Stride parameter for each layer
$W_{index} \leftarrow$ The allocated weight from output to x_{index}
Function $getWeight$
if $m == 1$ **then**
  for $index \in indices$ **do**
    $W_{index} \leftarrow W_{index} + \alpha^m$
  end for
else
  for $index \in indices$ **do**
    $newIndices \leftarrow []$
    for $i \in [1, K[m-1]]$ **do**
      $newIndex \leftarrow S[m-1] * (index - 1) + i$
      $nI.append(newIndex)$
    end for
    $\alpha^{m-1} \leftarrow \alpha^m / len(nI)$
    $getWeight(nI, \alpha^{m-1}, m-1, K, S, W_{indices})$
  end for
end if

Accumulation of multiple networks. As Figure 4 shows, we assume that the input $(x_1, x_2, ..., x_n)$ is sent to an m-layer network and the relevance score of $\vec{o}_j^m$ is $\vec{\alpha}$. As such, the relevance score of an input node x_i is $r_i = \sum_j W_i^{\vec{o}_j^m}$. Since $\vec{r}$ comes from $\vec{\alpha}$, whose length is shorter than $\vec{x}$, it is less accurate to indicate the importance of input node using $\vec{r}$. To this end, we take the average of multiple networks with different initial parameters to reduce errors. In particular, for a network p, we denote the input $\vec{x}$'s relevance

score as $\vec{r}^p = (r_1^p, r_2^p, ..., r_n^p)$. Then we normalize $\vec{r}^p$ and get $\vec{R}^p = (R_1^p, R_2^p, ..., R_n^p)$, where $R_i^p = r_i^p / max(\vec{r}^p)$. In the end, we accumulate multiple networks to calculate the final relevance score $\vec{R} = (R_1, R_2, ..., R_n)$.

4.5 Reduction

For a crashing input, its relevance score represents the contribution of each input byte to the crash. As such, the input bytes with higher relevance score should be preserved. However, the relevance score $\vec{R}$ of one single input is not representative because there are errors existed in the fitting of network and the weight computation, leading to inaccuracy. Therefore, we empirically select the top two percent of positive samples that are most different in the corpus (choosing the samples with the lowest Bi-gram scores) as the candidate set. Given that those samples are more likely to cover less explored paths in the program, there is a higher chance to reduce the mutated fields in the samples while triggering the crash. For each crashing input in the candidate set, we rank the input bytes according to the relevance score $\vec{R}$ and reduce them from high scores to lower scores. As described in Algorithm 3, binary search is used in our reduction. In addition, we execute the program to verify whether the reduced input indeed triggers the crash after reduction.

Algorithm 3 Reducing algorithm

Require: *crash_input* ← Crashing input to be reduced
R ← relevance score
sortedR ← *sort*(*R*)
reduceLen ← *len*(*crash_input*)/2
reducePos ← *len*(*crash_input*)/2
while *reduceLen* ∈ (1, *len*(*crash_input*)) **do**
  thd ← *sortedR*[*reducePos*]
  newinput ← ""
  for *i* ∈ (0, *len*(*crash_input*)) **do**
    if *sortedR*[*i*] > *thd* **then**
      newinput += *crash_input*[*i*]
    end if
  end for
  reduceLen /= 2
  if *Is_Crash_Triggered*(*newinput*) **then**
    reducePos −= *reduceLen*
  else
    reducePos += *reduceLen*
  end if
end while
return newinput

5 Evaluation

In this section, we describe the implementation, experiment setting, and the evaluation results. We also present the comparison between SCREAM and state-of-the-art solutions. In Appendix, we demonstrate two case studies to further illustrate how SCREAM accomplishes the reduction task.

5.1 Experimental Setting

Testing programs. In the experiments, we evaluated 41 programs including 29 CGC programs and 12 real-world programs. To fairly choose the programs, we select the programs that have known crashes[3] and belong to different software categories from the CGC program repositories and the CVE list, without examining the details of crash and program internals. The functionalities of CGC programs include gaming, image processing, audio decoding, video decoding, network protocol parsing, document file parsing, instruction emulation, router simulation, mail service and etc. The real-world programs are mainly used for image and document processing. In addition, the crashing inputs are produced with afl-fuzz[4].

Experimental setting. The experiments run on a Ubuntu 18.04 host machine with Intel i9-7900X CPU and 2080ti GPU. We use the platform TensorFlow with Keras version 2.1.1. For each program under test, multiple network instances are running with different parameters at the same time to reduce deviation. In particular, we train 10 network instances at the same time and each network is trained for 10 times. In the experiments, the fitting rates are found to be relatively high (larger than 90%) after the training, indicating that the trained networks are suitable for reducing test cases. Still, we select the most fitted iteration round to obtain relevance score *R* for the crashing input (the fitting rate is shown in Figure 4).

In regard to network hyper-parameters, since the convolutional layers are used to encode the input, the depths of convolutional layer and the kernel size are related to the size of program input. We empirically set the depth of the convolutional network based on the size of input and set the kernel size and the stride according to the network depth. Similarly, the size of feature vectors are also determined by the Conv1D parameters and are empirically set. To make sure that the attention mechanism can differentiate different input bytes and to prevent vanishing gradient problems in LSTM, the output size of the convolutional network is empirically set to a value from 30 to 120 (when setting the output size to, for example, 500, the LSTM is unable to handle the situation). Besides, we use L1 regularization to prevent overfitting.

Although the optimizations of hyper-parameters are important to the fitting, the effect after optimization is still case-by-case [11, 20]. Therefore, in practice, the hyper-parameters are often empirically set by analysts based on their experience. In that sense, some deviations are tolerable as long as the scale is suitable and performance is satisfying. During the experiments, we also tuned the hyper-parameters in different scales and determined a set of combinations of hyper-parameters that achieve satisfying performance. Table 3 shows a set of candidate Conv1D parameters that we used in the experiments.

[3] For real-world programs, the crashes are fixed in new versions.

[4] With sufficient time of fuzzing (one week), afl-fuzz identified crashes in 29 out of 40 CGC programs. However, we also selected 40 real-world programs and afl-fuzz only reproduced crashes in 12 of them.

The combinations of Conv1D parameters are chosen from the table with given input size.

Table 3: The convolutional layer's parameter setting

Input Size	Parameters of Conv1D (kernel, stride)	# of Layers
<500	(3,2),(3,1),(5,1),(5,2),(5,3)	3
<5000	(3,2),(3,1),(5,1),(5,2),(5,3),(5,4) (7,1),(7,2),(7,3),(7,4),(7,5),(7,6)	5
<50000	(3,2),(3,1),(5,1),(5,2),(5,3),(5,4) (7,2),(7,3),(7,4),(7,5),(7,6),(9,3) (9,4),(9,5),(9,6),(9,7),(9,8)	7

5.2 Overall Results

Table 4 in Appendix shows the overall statistics including program name, details of crash, reduction rate (i.e., $(size(I_{crashing}) - size(I_{result}))/size(I_{crashing})$), time cost, as well as the comparison on reduction, dataset augmentation, and interpretability. The evaluated programs are crashed due to variant causes, such as stack-based buffer overflow, heap-based buffer overflow, out-of-bounds read and write, integer overflow and etc. The size of crashing input varies from tens of bytes to hundreds of kilobytes. Besides, given that the fuzzer produces multiple crashing inputs for each program's crashing point, we evaluated two randomly-selected crashing inputs per crash in the experiment. One crashing input per crash is considered as a case. As such, there are 82 cases in total.

Reduction rate and time cost. On the whole, SCREAM achieves an average reduction rate of 75.4% which takes 29.8 minutes on average (the training time of multiple networks is accumulated). The achieved reduction rate is highly related to the size of crashing input and the ground truth but not necessarily related to the root cause of crash, as shown in the statistics of Table 4. In terms of time consumption, time is mostly spent on training networks. For the network training, since we train multiple networks at the same time, it takes around 20 to 90 seconds for the training of one round for one network. For relevance score computing, it takes around 5 to 20 seconds for each program.

Comparison with the state-of-the-art solutions. We compared SCREAM with afl-tmin [1], Picireny [4], and Penumbra [21]. Afl-tmin is a widely-used test case minimizer integrated with American Fuzzy Lop (afl). It is a prominent example of delta debugging implementation. Picireny is a open-source hierarchical delta debugging framework, which makes use of pre-defined structures of inputs to improve delta debugging [28, 29]. Penumbra leverages dynamic taint analysis to identify failure-relevant inputs.

We noticed that using extra time cost and extra reduction rate to compare SCREAM with other tools is imprecise. For instance, taking extra 15 minutes to achieve an extra reduction rate of 30% does not mean that SCREAM is more or less efficient than the other tool. The trade-offs do not just exist between time and reduction rate. Other factors such as accuracy of reduction also matter. To this end, we define the reduction efficiency E in Equation 8, which takes the amount of reduction, time cost, and accuracy of reduction into consideration. We use *relative efficiency* R in Equation 9 to fairly compare the reduction efficiency, where:

$$E = \frac{size(I_{crashing}) - size(I_{result})}{time\ cost} \times \frac{size(I_{minimal})}{size(I_{result})} \quad (8)$$

$$R = \frac{E_{SCREAM}}{E_{a\ tool}} \quad (9)$$

In Equation 8, $size(I_{minimal})\ /size(I_{result})$ measures the accuracy of reduction[5], where $I_{minimal}$ is the ground-truth input and I_{result} is the resulting input after reduction. $(size(I_{crashing}) - size(I_{result}))\ /time\ cost$ is the amount of reduction achieved by the tool in a period of time, where $I_{crashing}$ is the crashing input.

The relative efficiency for afl-tmin, Picireny and Penumbra on all evaluated programs is show in Figure 6. When R is larger than 1 (i.e., $logR$ is larger than 0), SCREAM outperforms the other tool. For instance, in case 12-1, the size of the crashing input is 231, and the ground truth is 60. SCREAM reduced the input size to 60 taking 15 minutes, and afl-tmin reduced the input size to 178 taking 3 minutes. As a result, the calculated reduction efficiency is 11.4 for SCREAM and 5.9 for afl-tmin. The efficiency ratio is 1.91, meaning that SCREAM is more efficient. Although SCREAM takes more time, such extra time consumption is worthwhile considering the extra reduction.

When compared with afl-tmin and Penumbra, SCREAM demonstrates surprising reduction capability with negligible extra overhead, by achieving an extra reduction rate of 29.7% and 53.4% with extra 12.2 minutes and 11.42 minutes, respectively. For the comparison with Picireny, SCREAM's extra reduction rate is 29.7%, and the average time cost is 4.75 minutes less. Furthermore, SCREAM achieves a higher reduction efficiency on 86.5% cases, 89.1% cases, and 100% cases, when compared with afl-tmin, Picireny, and Penumbra, respectively (shown in Figure 6). The low reduction efficiency of afl-tmin can be attributed to the fact that the crashing inputs contain dependent and discontinuous blocks which increase the iteration round for afl-tmin. For Picireny, since no input structure is given for the general programs in our evaluation, it is depreciated to a delta debugging tool written in Python with no optimizations. For Penumbra, the reduction efficiency largely relies on the accurate specification of taint sinks which involves root cause analysis of crash. In practice, however, the specification is inevitably inaccurate for different types

[5]This value is small if the false negative rate is large, meaning that there is still much reduction work left to do.

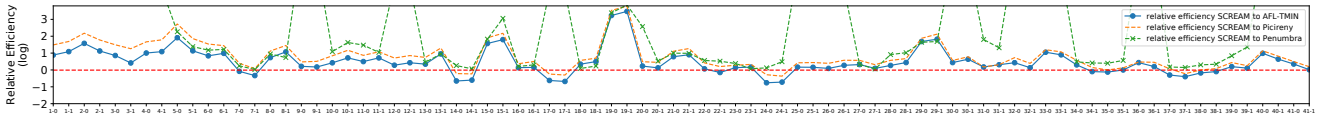

Figure 6: Relative efficiency of SCREAM, afl-tmin, Picireny and Penumbra

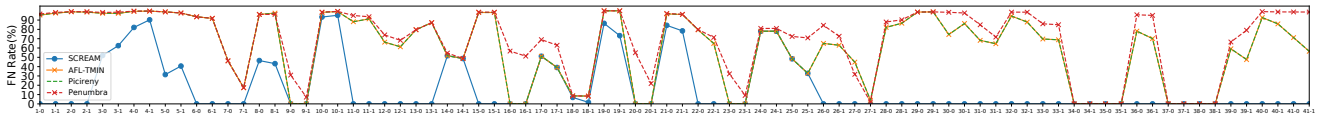

Figure 7: False negative rate of SCREAM, afl-tmin, Picireny and Penumbra

of crashes, resulting in imprecise information flows and low reduction efficiency in the experiments.

Trade-offs between training time and performance. We found that the effectiveness of reduction can be significantly affected if the network training is not finished or the algorithms do not terminate. For instance, when the training time is only a half of the normal training time, we use the unfitted network to identify failure-inducing inputs and the average reduction rate is only 29.2%. Therefore, to complete the reduction task, all the tools should be sufficiently run to reach their bottleneck. In the experiments, we make sure that (1) afl-tmin, Picireny and Penumbra are sufficiently run and properly exit; (2) the neural network is well trained according to the criteria described in Section 4 before it works on the reduction.

5.3 Accuracy and Generality

False positives and false negatives. As described in Section 4.5, our approach adopts binary search to reduce irrelevant input bytes and execute the program to confirm whether the crash is indeed triggered. Therefore, SCREAM does not report false positives. To check whether there are any false negatives, we manually constructed minimal failure-inducing inputs for each program crash after an automated reduction procedure (i.e., by manually examining the reduced inputs and digging into program logic to remove irrelevant bytes). There is no guarantee that global optimum of reduction is achieved. Thanks to SCREAM's capability to continuously mutate inputs and progressively reduce them by adjusting the significance, the average false negative rate ($FNR = (size(I_{crashing}) - size(I_{minimal}))/size(I_{crashing})$) of SCREAM is 17.0% while the average FNR of afl-tmin, Picireny and Penumbra is 60.8%, 60.8% and 69.5%, respectively (Figure 7 shows the FNR of SCREAM, afl-tmin, Picireny and Penumbra on each case). With the help of SCREAM, 70.7% of the reduced inputs have reached ground truth. Such improvement is made possible by SCREAM's capability to summarize the program logic, like transformation of control flow-led input bytes, limitation on input length, requirement of specific format, and etc.

Benefits of SCREAM**.** Although the size crashing input could be extremely large in some cases, SCREAM is able to focus on a subset of input bytes that are neither significant nor insignificant and continuously mutate them, in order to make a deviation from existing crashing inputs and achieve control flow transfer in the program to some extent. From the evaluation, we found that SCREAM is able to solve control flow complexity in the following cases:

- The crashing input contains multiple discontinuous input blocks that must be reduced at the same time. This kind of constraint on input bytes appears in string search programs, chat programs, database programs, image processing programs, news feed programs, calendar programs and etc.
- The crashing input contains input blocks with specific format (e.g., the format of IP address). We found that such a constraint appears in software such as string search, route management, 3D maps, instruction emulator, json parser, photo management, image processing, mail service client, TCP protocol stack and etc.
- The crashing input contains a field that specifies the minimal input length. The constraint appears in software types like image processing, file compressing, string search, profile management, Bluetooth communication management and etc.
- The crashing input contains input blocks that directly affect the program's control flow. This constraint appears in instruction emulators, calculators, document format converter, and document processing programs. The above constraints are common in various types of software, and it is easy for SCREAM to produce inputs that satisfy them by marking significance on the input bytes.

Limitations of SCREAM**.** Nevertheless, false negatives occur in the presence of complex arithmetic operations such as checksum and other checksum-like functions that involve calculation and multiple exact match of data values. In our evaluation, the complex arithmetic operations are shown in the software that involves error detection, data integrity check

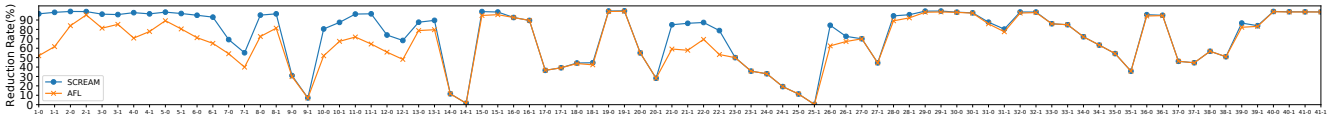

Figure 8: Reduction rate when adopting SCREAM's mutation technique and AFL's mutation engine

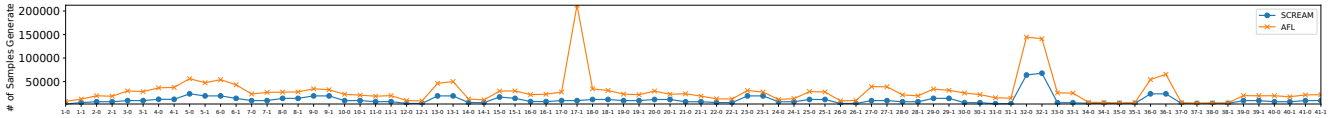

Figure 9: # of Samples generated by SCREAM's mutation technique and AFL's mutation engine

and data indexing, such as audio processing software, video processing software, and USB communication management (packet processing) software.

In the presence of complex arithmetic operations in program inputs, the reduction efficiency of afl-tmin and Picireny is higher than that of SCREAM. Given the simplicity of afl-tmin and Picireny's reduction algorithm, they are unable to handle the complexity in input structures as well. In this case, afl-tmin and Picireny take less time in processing the inputs, which results in higher reduction efficiency.

Limited by current computation power and network architectures, handling complex arithmetic operations like checksum-like functions is still beyond the expressiveness of neural networks.

5.4 Comparison on Dataset Augmentation and Interpretability

Comparison with AFL's mutation engine. To demonstrate the efficacy of our dataset augmentation algorithm, we compared our approach with AFL's mutation engine [49]. To construct a dataset with the samples produced by AFL, we modified AFL by removing its path exploration functions and preserving its genetic algorithm-based mutation functions. As the neural network takes input with a fixed size, we select the samples of which the size is equal to or smaller than the original crashing input. For the produced samples with shorter sizes, we fill the gap with "-1". Therefore, all the samples produced by AFL have the same size as the size of the original input. After that, we send the dataset constructed by AFL's engine to the backend (i.e., the network component) of SCREAM for further processing. It turns out that when using AFL's mutation engine, more samples are produced (on average, the number of samples consumed is increased by 163.4% as shown in Figure 9.) but less reduction efficiency is achieved (shown in Figure 8). Thanks to SCREAM's dataset augmentation algorithm which tends to produce highly-differentiable samples in input space, it is easier for the neural network to draw a more accurate boundary even with less samples. On the other hand, with our approach, each round of calculated relevance score can correct the deviation during network training, which tends to produce a more accurate relevance score in the end.

Comparison with partial derivatives. Partial derivatives is also an important approach of interpretability which has been used in binary analysis tasks (e.g., *Neuzz* [41] and *NeuTaint* [40]]). To compare the two interpretability approaches, namely the attention mechanism and partial derivatives, we use the implementation of Neuzz with the same dataset to calculate the significance. Figure 10 shows the reduction efficiency when adopting both approaches. It turns out that less reduction efficiency is achieved by the partial derivative approach. The main cause is that, there exists the saturation problem for partial-derivative-based approach, which underestimates the importance of features to the output and affects the calculated significance. In the experiments, when the input length is 200, SCREAM and neuzz have similar magnitude of trainable parameters. Nevertheless, when the input length reaches 1000, the trainable parameters of neuzz are at least ten times of SCREAM's, which causes overfitting to the network. This eventually affects the calculation on the significance.

6 Related Work

6.1 Test Case Reduction

Existing techniques of test case reduction can be categorized as random reduction and rule-based approaches. Random reduction approaches treat the program as a blackbox and randomly or strategically mutate the program input. A prominent example is delta debugging. Zeller et al. [50] proposed to use binary search for delta debugging. The core idea is to randomly reduce a portion of input and gradually increase the granularity of reduction. This approach assumes that the input bytes are independent with each other. Otherwise, the reduction of dependent input bytes could lead to failures. As a result, delta debugging is not applicable to file (e.g., documents, images, and etc.) processing programs, as those programs digest structural inputs that usually involve interdependence among input blocks. To this end, Groce et al. [25], Regehr et al. [37],

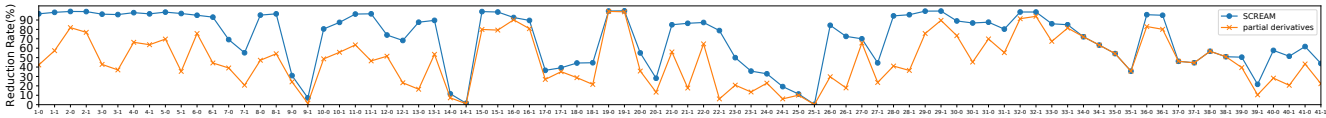

Figure 10: Reduction rate when adopting SCREAM's attention mechanism and partial derivatives

and Pike et al. [36] proposed different reduction strategies to target specific programs with known input structures. However, those approaches are less adaptable to general programs. On the other hand, in regard to rule-based approaches, Clause and Orso [21] proposed to mark the input bytes that contribute to the crash using taint analysis. However, this requires manual analysis of the crash to determine taint sinks. Moreover, making accurate taint policy is challenging because of the control flow dependence problem.

Another line of research is root cause analysis. To locate the program entities (e.g., functions and basic blocks) related to the crashing point, existing root cause analysis approaches leverage statistical information of program runtime behaviors [6,12,26,47], use backward dataflow analysis [7,39], or utilize information from bug reports [45, 46, 52]. Since root cause analysis is complicated and still involves manual analysis, it cannot be used for test case reduction. In contrast, test case reduction is a necessary step to improve the accuracy of identifying a candidate set of crash-related program entities.

The problem of test case reduction is also related to fuzzing [13, 18, 24, 35, 41, 43]. The purpose of test case reduction is to reduce the length of crashing inputs after crashes have been triggered in fuzzing. To this aim, SCREAM produces diverse inputs that are scattered in the input space, for the purpose of network training and reduction. Different from that, fuzzers like AFL [49], Driller [43], Angora [18] and Neuzz [41] tend to produce inputs that are clustered in the input space, with the aim of exploring program paths and triggering/discovering crashes. The way of fuzzers' input generation is not suitable for neural-network-based test case reduction.

6.2 Interpretability of Neural Networks

With proper datasets, the neural network can automatically fit the function of the input and the output. The interpretability of neural network is to understand how each input component affects the output, and the method that explores every input component's influence to the output is called network explanation method. Omeiza et al. [34] proposed to determine the importance of every input pixel to the output by computing the gradient value of a fitted neural network. However, it would lead to the saturation problem which underestimates the importance of features to the output. Shrikumar et al. [42] compared the activation of each neuron to its "reference activation" and assigned relevance scores according to the difference. Such a method is applicable to the explanation of the network itself and cannot compute the relevance score for the input. Sutskever et al. [44] proposed the attention mechanism with seq2seq networks by distributing the weight to each input component using a similarity function, which is supposed to avoid the saturation problem. After that, the attention mechanism has been widely applied in the NLP area. Generally, the explanation method of neural network aims to improve the fitting accuracy of the network. In this paper, we utilize the explanation method to determine the failure-inducing input that contributes to the crash.

7 Conclusion

In this paper, we have presented SCREAM, a deep learning-based solution for test case reduction. In particular, we utilize the neural network to approximate the computation from the program input to the crash and leverage the attention mechanism of neural network to determine the contribution of each input bytes to the crash. We also presented several novel techniques including an online dataset augmentation technique that can produce highly-differentiable samples and works in conjunction with the network, and a new network architecture to process long input sequences. We evaluated SCREAM on 41 programs including 29 CGC programs and 12 real-world programs. The results show that our approach is effective and accurate in test case reduction.

References

[1] " afl-tmin - test case minimizer for American Fuzzy Lop (afl)," http://manpages.ubuntu.com/manpages/xenial/man1/afl-tmin.1.html, Accessed: May 2021.

[2] "Lecture Notes on Delta Debugging," https://www.cs.purdue.edu/homes/suresh/408-Spring2017/Lecture-9.pdf, Accessed: May 2021.

[3] "On-Hot - Wikipedia," https://en.wikipedia.org/wiki/One-hot, Accessed: May 2021.

[4] "Picireny: Hierarchical Delta Debugging Framework," https://github.com/renatahodovan/picireny, Accessed: May 2021.

[5] "Up-sampling," https://en.wikipedia.org/wiki/Upsampling, Accessed: May 2021.

[6] R. Abreu, P. Zoeteweij, and A. J. C. V. Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques-mutation*, 2007.

[7] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, "Fault localization using execution slices and dataflow tests," in *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95*, 2002.

[8] C. Artho, "Iterative delta debugging," *International Journal on Software Tools for Technology Transfer*, vol. 13, no. 3, pp. 223–246, 2011.

[9] P. Arumuga Nainar, T. Chen, J. Rosin, and B. Liblit, "Statistical debugging using compound boolean predicates," in *Proceedings of the 2007 international symposium on Software testing and analysis*, 2007, pp. 5–15.

[10] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[11] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *J. Mach. Learn. Res.*, vol. 13, pp. 281–305, 2012.

[12] 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 *29th USENIX Security Symposium*, 2020.

[13] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2329–2344.

[14] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2017.

[15] J. Buckman, A. Roy, C. Raffel, and I. J. Goodfellow, "Thermometer encoding: One hot way to resist adversarial examples," in *ICLR*, 2018.

[16] M. Carbin and M. C. Rinard, "Automatically identifying critical input regions and code in applications," in *Proceedings of the 19th international symposium on Software testing and analysis*, 2010, pp. 37–48.

[17] S. Chaudhari, G. Polatkan, R. Ramanath, and V. Mithal, "An attentive survey of attention models," *arXiv preprint arXiv:1904.02874*, 2019.

[18] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*.

[19] A. Christi, M. L. Olson, M. A. Alipour, and A. Groce, "Reduce before you localize: Delta-debugging and spectrum-based fault localization," in *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2018, pp. 184–191.

[20] M. Claesen and B. D. Moor, "Hyperparameter search in machine learning," *CoRR*, vol. abs/1502.02127, 2015.

[21] J. Clause and A. Orso, "Penumbra: automatically identifying failure-relevant inputs using dynamic tainting," in *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 2009, pp. 249–260.

[22] H. Cleve and A. Zeller, "Locating causes of program failures," in *27th international conference on Software Engineering*. ACM, 2005.

[23] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008, pp. 206–215.

[24] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: whitebox fuzzing for security testing," *Queue*, vol. 10, no. 1, pp. 20–27, 2012.

[25] A. Groce, G. Holzmann, and R. Joshi, "Randomized differential testing as a prelude to formal verification," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 621–631.

[26] M. J. Harrold, G. Rothermel, and K. Sayre, "An empirical investigation of the relationship between spectra differences and regression faults," *Software Testing, Verification and Reliability*, 2000.

[27] X. He, Z. He, J. Song, Z. Liu, Y.-G. Jiang, and T.-S. Chua, "Nais: Neural attentive item similarity model for recommendation," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 12, pp. 2354–2366, 2018.

[28] R. Hodován and Á. Kiss, "Modernizing hierarchical delta debugging," in *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation, A-TEST@SIGSOFT FSE 2016, Seattle, WA, USA, November 18, 2016*, T. E. J. Vos, S. Eldh, and W. Prasetya, Eds.

[29] R. Hodován, Á. Kiss, and T. Gyimóthy, "Tree preprocessing and test outcome caching for efficient hierarchical delta debugging," in *12th IEEE/ACM International Workshop on Automation of Software Testing, AST@ICSE 2017, Buenos Aires, Argentina, May 20-21, 2017*.

[30] W. Jin and A. Orso, "F3: fault localization for field failures," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*.

[31] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2123–2138.

[32] J. Li, W. Monroe, and D. Jurafsky, "Understanding neural networks through representation erasure," *arXiv preprint arXiv:1612.08220*, 2016.

[33] G. Misherghi and Z. Su, "HDD: hierarchical delta debugging," in *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, L. J. Osterweil, H. D. Rombach, and M. L. Soffa, Eds. ACM, pp. 142–151.

[34] D. Omeiza, S. Speakman, C. Cintas, and K. Weldermariam, "Smooth grad-cam++: An enhanced inference level visualization technique for deep convolutional neural network models," *arXiv preprint arXiv:1908.01224*, 2019.

[35] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: Fuzzing by program transformation," in *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*.

[36] L. Pike, "Smartcheck: automatic and efficient counterexample reduction and generalization," in *ACM SIGPLAN Notices*, vol. 49, no. 12. ACM, 2014, pp. 53–64.

[37] J. Regehr, Y. Chen, P. Cuoq, E. Eide, and C. Ellison, "Test-case reduction for c compiler bugs," in *ACM SIGPLAN Notices*, 2012.

[38] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for C compiler bugs," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, J. Vitek, H. Lin, and F. Tip, Eds. ACM, 2012, pp. 335–346.

[39] M. Renieres and S. P. Reiss, "Fault localization with nearest neighbor queries," in *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.* IEEE, 2003, pp. 30–39.

[40] D. She, Y. Chen, A. Shah, B. Ray, and S. Jana, "Neutaint: Efficient dynamic taint analysis with neural networks," in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1527–1543.

[41] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "NEUZZ: efficient fuzzing with neural program smoothing," in *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*.

[42] A. Shrikumar, P. Greenside, and A. Kundaje, "Learning important features through propagating activation differences," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2017, pp. 3145–3153.

[43] 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, no. 2016, 2016, pp. 1–16.

[44] I. Sutskever, O. Vinyals, and Q. Le, "Sequence to sequence learning with neural networks," *Advances in NIPS*, 2014.

[45] C. P. Wong, Y. Xiong, H. Zhang, D. Hao, and H. Mei, "Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis," in *IEEE International Conference on Software Maintenance and Evolution*, 2014.

[46] R. Wu, H. Zhang, S.-C. Cheung, and S. Kim, "Crashlocator: locating crashing faults based on crash stacks," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 204–214.

[47] X. Xie, T. Y. Chen, F. C. Kuo, and B. Xu, "A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization," *Acm Transactions on Software Engineering and Methodology*, vol. 22, no. 4, pp. 1–40, 2013.

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

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

[50] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[51] X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 272–281.

[52] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *International Conference on Software Engineering*, 2012.

Appendices

A Case Studies

Jhead. Jhead is a real-world command-line tool for processing EXIF information of images. The EXIF information includes camera model, resolution, ISO value, GPS information, and etc. When the jhead program parses the segment that stores GPS information, providing a negative float value in the input would cause a stack-based overflow in the `sprintf` function at line 17, as the program does not check the sign of the variable.

```
1   void ProcessGpsInfo(unsigned char *
        DirStart,
2   unsigned char *OffsetBase, unsigned
        ExifLength)
3   {...
4   ValuePtr = OffsetBase+OffsetVal;
5   ...
6   switch(Tag){
7   char TempString[50];
8   ...
9   case TAG\_GPS\_LAT:
10  case TAG\_GPS\_LONG:
11  ...
12  strcpy(FmtString,"%0.0fd %0.0fm %0.0fs");
13  ...
14  for (a=0;a<3;a++){
15  int den, digits;
16  Values[a] = ConvertAnyFormat(ValuePtr+a*
        ComponentSize, Format); }
17  sprintf(TempString, FmtString, Values[0],
        Values[1], Values[2]);
18  ...
19  break;
20  ...}}
```

Listing 2: The code where the fault locates

The important input bytes that contribute to the crash are the control data "`ff 05 00`" that directs the program to process GPS information and the invalid float value that causes the overflow. Taint analysis approaches cannot identify the input bytes that contribute to the crash, as the invalid float value is from `snprintf` function which is not directly from the program input.

In Figure 11 (horizontal axis represents index of input byte, while vertical axis represents importance), we show how SCREAM scores the importance of the input bytes in round 1, round 5, and round 10. As can be seen, the score of the input bytes that are irrelevant to the crash decreases during the training process. Initially the original size of crashing input is 320 bytes. At round 5, the relevance score becomes stable and SCREAM outputs a failure-inducing input with 63 bytes. The mutation of samples and the network training is based on the result of previous rounds, which allows the network to correct errors in the fitting. In this case, the 320-byte input size is reduced to 63 bytes in 5-minute training.

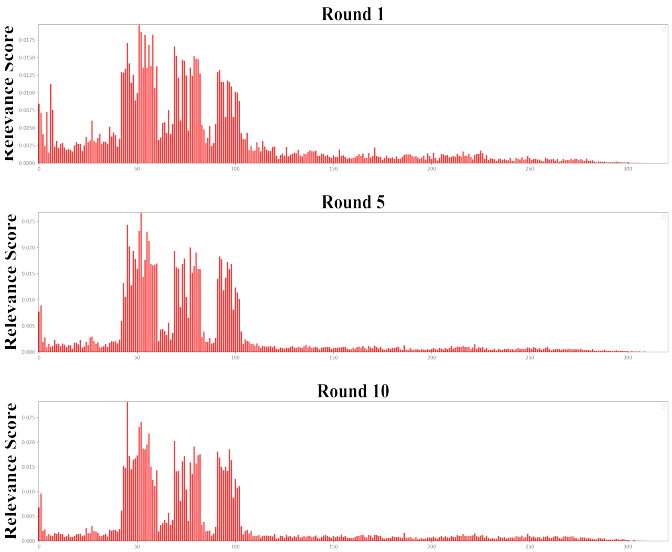

Figure 11: Relevance score of jhead at round 1, 5 and 10

Simple stack machine. Simple Stack Machine is a CGC program and an instruction emulator that takes instructions as input. This program requests a piece of heap area and uses it as the "stack" of the emulator. The crash was discovered using AFL. We show the crashing input in Listing 3. As can be seen, starting from line 1 to line 6, a series of "`push`" instructions push data to the stack. At line 7, a "`sub`" instruction stores the subtracted result "0" to the stack. At line 8, the "`jmpz`" instruction pops the top value off of the stack. If the value is 0, the emulator pops the next value off the stack and uses it as the next instruction. As such, the root cause of the crash is that, as long as the top of stack stores two "0"s, the emulator will jump to the first byte of the input at line 1, which forms a dead loop, keeps pushing data to the stack, and eventually causes heap overflows.

```
1   00 41 33 60 //push 0xC066820
2   ......
3   00 32 d3 d3 // push 0x1A7A6640
4   00 00 00 00 // push 0x0
5   00 04 00 02 // push 0x400080
6   00 04 00 02 // push 0x400080
7   7f 12 39 d1 // sub; pop top two on the
        stack and push the result to the stack
8   d3 c6 d3 d3 //jmpz
9   00 00 00 7f // push 0xFE00000
10  ff ff ff ff // end
11  12 34 12 35 12 35 12 45
```

Listing 3: The crashing input in round 1

Actually, the push instructions from line 1 to line 3 are irrelevant to the crash. For afl-tmin, to trigger the crash, it has to keep the instructions from line 4 to line 8. For our approach, in the first round, SCREAM outputs a preliminary relevance score indicating that instructions from line 4 to line 8 are significant, which is the same as the afl-tmin's result.

As the "`push 0x0`" instruction at line 4 receives the highest score (mutating it would not lead to any crashes), the next round of mutation only acts on the instructions from line 5 to line 8. We show the intermediate result in round 7 in Listing 4. After that, the iteration proceeds until round 10 when the relevance score becomes stable (the change of relevance score is shown in Figure 12). In the end, a final failure-inducing input is produced in Listing 5. In this example, the input size is reduced from 73 bytes to 20 bytes in 8-minute training

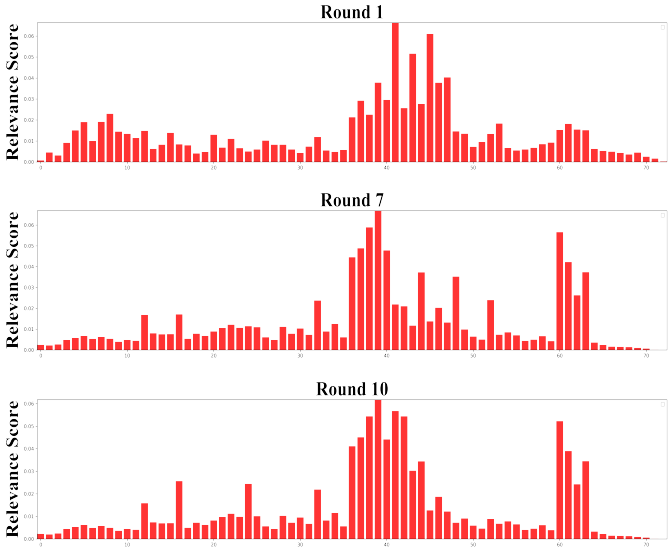

Figure 12: Relevance score of Simple Stack Machine at round 1, 7 and 10

```
1   80 ff 33 60 // push 0xC067FF0
2   a8 fd fa 91 // push 0x123F5FB5
3   ......
4   f0 32 f0 d3 // push
5   00 00 00 00 // push 0x0
6   00 04 00 02 // push 0x400080
7   f1 04 00 02 // pop
8   d3 57 27 28 // jmpz
9   d3 ff b2 4c // jmpz
10  12 ba de f5 // pushpc, push 0xC
11  ff ff ff ff // end
12  12 10 56 1c 85 1c dd 36
```

Listing 4: The crashing input generated in round 7

```
1   80 ff 33 60 // push 0xC067FF0
2   ......
3   f0 32 f0 d3 // push 0x1A7E065E
4   00 00 00 00 // push 0x0
5   00 00 00 00 // push 0x0
6   f7 ff d8 ad // jmpz
7   09 89 67 13 // pushpc,push 0xB
8   46 57 42 ed // add
9   f2 90 76 36 // pushpc,push 0xD
10  ff ff ff ff // end
11  12 10 56 1c 85 1c dd 36
```

Listing 5: The crashing input generated in round 10

Table 4: Overall Statistics

No	Program	Size of Crash Input	Ground Truth Input	Fitting Rate (%)	Reduction Rate (%)	Time (min)	# of Samples (Pos. / Total)	Comparison on Reduction (w/ afl-tmin/Picirency/Penumbra)		Comparison on Dataset Augmentation (w/ AFL)		Comparison on Interpretability (w/ Partial Derivatives)	
								Time (min)	Relative Efficiency	# of Samples (Pos|Total)	Relative Efficiency	Reduction Rate (%)	Relative Efficiency
1	String Storage and Retrieval	348/648	12/12	(95.0|99.0)/(94.0|99.0)	96.6/98.1	21/24	(869|2400)/(1896|5400)	(2/8/8)/(2/8/8)	(2.1/0.5/N/A)/(2.2/0.5/N/A)	(2785/7861)/(4243/12716)	84.6/77.4	40.0/60.0	21.9/31.1
2	TextSearch	542/476	5/5	(93.0|98.0)/(95.0|99.0)	99.1/98.9	32/38	(2593|7200)/(2637|7200)	(3/12/7)/(2/9/9)	(0.4/0.1/N/A)/(0.9/0.2/N/A)	(6746/19871)/(6807/18927)	56.7/12.0	80.0/80.0	12.6/11.2
3	stream_vm	659/758	12/12	(92.0|97.0)/(95.0|98.0)	96.2/95.8	21/22	(3475|9600)/(3236|9600)	(3/13/12)/(2/14/11)	(2.0/0.5/N/A)/(4.8/0.7/N/A)	(10865/30182)/(10423/28719)	18.3/11.6	40.0/40.0	30.7/35.2
4	stack_vm	3050/3550	12/12	(94.0|99.0)/(95.0|99.0)	97.8/96.5	48/50	(4488|12400)/(4243|12400)	(5/23/23)/(5/25/22)	(1.1/0.2/N/A)/(0.5/0.1/N/A)	(13223/36716)/(12906/37612)	54.4/24.2	70.0/60.0	15.1/11.1
5	humaninterface	8762/5341	98/98	(91.0|97.0)/(93.0|96.0)	98.4/96.9	68/66	(8526|24000)/(6857|19600)	(8/52/18)/(9/48/16)	(1.1/0.2/0.5)/(3.4/0.6/1.9)	(20107/56271)/(16072/47561)	17.0/18.5	70.0/40.0	21.2/36.5
6	SFTSCBSISS	3271/2331	162/162	(94.0|98.0)/(95.0|97.0)	95.0/93.1	53/56	(7119|19600)/(4872|14400)	(6/30/13)/(7/20/12)	(8.5/1.7/3.9)/(4.0/1.4/2.3)	(19549/54212)/(14388/43121)	21.5/21.4	80.0/40.0	4.4/9.6
7	router_simulator	3461/2381	1067/1067	(95.0|99.0)/(95.0|98.0)	69.2/55.2	29/22	(3365|9600)/(3325|9600)	(8/24/17)/(7/18/16)	(99.4/33.1/46.8)/(128.7/50.0/56.3)	(8296/23612)/(9545/27152)	4.7/5.2	40.0/20.0	4.5/6.2
8	Resort_Modeller	3584/4673	92/92	(94.0|96.0)/(93.0|99.0)	95.2/96.5	76/82	(5111|14400)/(5158|14400)	(11/27/18)/(10/24/16)	(4.4/1.8/2.7)/(2.7/1.1/5.7)	(9912/27612)/(9916/28089)	14.5/12.5	50.0/50.0	10.2/12.1
9	reallystream	189761/141291	131103/131103	(95.0|97.0)/(92.0|98.0)	30.9/7.2	44/38	(6583|19600)/(7096|19600)	(72/132/26)/(57/123/23)	(814.7/444.4/N/A)/(178.7/82.8/N/A)	(12117/34761)/(11756/32677)	1.9/1.7	20.0/0.0	4.2/17.0
10	Query_Calculator	456/982	6/6	(95.0|99.0)/(94.0|96.0)	80.5/87.5	28/18	(3328|9600)/(3328|9600)	(5/13/11)/(6/17/13)	(0.3/0.1/0.1)/(0.4/0.2/0.1)	(8362/23095)/(7595/21102)	9.2/7.5	50.0/60.0	3.6/11.4
11	PTaaS	787/847	29/29	(95.0|99.0)/(91.0|98.0)	96.3/96.6	34/35	(2490|7400)/(2682|7400)	(9/21/15)/(10/23/12)	(7.1/3.0/0.8)/(4.5/2.0/2.1)	(6633/19016)/(7377/20184)	26.2/42.4	60.0/50.0	15.9/41.7
12	PRU	231/189	60/60	(93.0|97.0)/(94.0|99.0)	74.0/68.3	15/11	(1217|3600)/(1289|3600)	(3/8/11)/(3/8/12)	(6.0/2.2/N/A)/(4.4/1.6/N/A)	(3510/9769)/(3253/8877)	6.1/5.7	50.0/20.0	2.5/8.4
13	Printer	4236/5034	523/523	(95.0|99.0)/(94.0|98.0)	87.7/89.6	69/72	(6901|19600)/(7089|19600)	(14/35/18)/(17/38/17)	(24.3/9.7/18.9)/(7.0/3.1/7.0)	(16641/46152)/(17850/50098)	4.5/5.6	20.0/50.0	24.6/5.8
14	Personal_Fitness_Manager	234/198	100/100	(95.0|99.0)/(93.0|98.0)	11.5/1.5	18/12	(1845|5400)/(1969|5400)	(4/11/16)/(3/7/14)	(3.3/1.2/0.4)/(0.5/0.2/0.1)	(4537/12981)/(3953/11077)	2.4/2.1	10.0/0.0	1.1/1.5
15	Overflow_Parkng	7234/5098	77/77	(93.0|96.0)/(95.0|98.0)	98.9/98.5	72/68	(6203|17200)/(4830|14400)	(22/48/20)/(16/38/16)	(2.7/1.2/1.5)/(1.2/0.5/0.1)	(10221/29888)/(10435/30198)	8.8/6.2	80.0/80.0	18.1/10.5
16	oneline_job_applicaton2	5781/4067	425/425	(92.0|99.0)/(95.0|97.0)	92.6/89.6	27/21	(2743|7600)/(2619|7600)	(37/63/17)/(30/67/18)	(144.8/85.0/122.3)/(121.4/54.4/86.5)	(7935/22312)/(8285/23123)	2.9/3.0	90.0/80.0	2.4/3.6
17	netstorage	548/459	170/170	(94.0|98.0)/(93.0|97.0)	36.5/39.2	21/24	(3208|9600)/(3240|9600)	(5/12/12)/(5/12/11)	(19.5/8.1/N/A)/(21.9/9.1/N/A)	(9291/27811)/(73085/212371)	2.9/22.1	30.0/40.0	1.7/0.9
18	Multi_User_Calendar	2509/2387	1301/1301	(93.0|96.0)/(93.0|98.0)	44.3/44.6	34/29	(4059|12000)/(4330|12000)	(72/120/40)/(77/123/42)	(13.8/8.3/24.8)/(11.5/7.2/21.1)	(11782/34617)/(11060/31076)	3.0/2.8	30.0/20.0	2.3/3.6
19	middleOut	102112/98351	62/62	(94.0|97.0)/(95.0|97.0)	99.6/99.8	42/42	(3401|9600)/(3263|9600)	(80/161/50)/(60/121/48)	(0.2/0.1/0.1)/(0.2/0.1/0.1)	(7848/23419)/(7686/22134)	5.3/6.6	100.0/100.0	4.6/12.9
20	Message_Service	5712/3572	2569/2569	(94.0|96.0)/(96.0|97.0)	55.0/28.1	46/39	(4291|12000)/(4393|12000)	(78/142/32)/(52/110/29)	(40.3/22.1/0.2)/(19.3/9.1/7.7)	(10690/29871)/(8291/23111)	2.5/1.9	40.0/10.0	2.3/2.4
21	LazyCalc	678/548	16/16	(94.0|97.0)/(93.0|95.0)	85.0/86.5	42/32	(2460|7200)/(2550|7200)	(20/42/17)/(15/35/16)	(0.4/0.2/0.2)/(0.4/0.2/0.3)	(8265/24211)/(6599/19081)	13.1/12.4	60.0/20.0	1.8/14.9
22	INSULATR	356/212	45/45	(95.0|98.0)/(92.0|94.0)	87.4/78.8	38/32	(1906|5400)/(1975|5400)	(4/9/12)/(4/9/10)	(7.0/3.1/2.2)/(7.4/3.3/1.6)	(4479/13411)/(4520/12987)	7.6/7.8	60.0/10.0	1.5/21.3
23	HIGHCOO	16382/12737	8194/8194	(95.0|98.0)/(91.0|94.0)	50.0/35.7	70/59	(6902|19600)/(6716|19600)	(96/134/60)/(87/126/52)	(85.3/61.1/48.5)/(52.2/36.1/65.2)	(10948/31009)/(10048/27871)	1.6/1.4	20.0/10.0	6.3/5.9
24	Audio_Visullizer	332/276	49/49	(95.0|99.0)/(91.0|93.0)	32.8/19.2	23/21	(2613|7200)/(2474|7200)	(4/12/18)/(4/9/19)	(6.0/2.0/0.8)/(2.9/1.3/0.2)	(4012/11982)/(4793/13811)	1.7/1.9	20.0/10.0	1.0/1.9
25	CGC_Video_Format_Parser_and_Viewer	14786/13987	4078/4078	(93.0|99.0)/(94.0|98.0)	46.6/56.6	49/48	(4331|12000)/(4145|12000)	(72/130/22)/(69/132/13)	(49.4/27.3/N/A)/(77.2/40.3/N/A)	(10098/28781)/(9576/27988)	2.4/2.3	10.0/0.0	1.0/2.5
26	Simple_Stack_Machine	128/73	20/20	(93.0|97.0)/(95.0|98.0)	84.4/72.6	7/8	(1308|3600)/(1306|3600)	(2/4/8)/(2/4/7)	(12.5/6.2/N/A)/(3.5/1.8/N/A)	(3079/9087)/(3406/9801)	8.2/3.5	30.0/20.0	18.3/12.2
27	basic_messaging	4321/2331	1295/1295	(95.0|97.0)/(94.0|98.0)	70.0/44.4	24/23	(3308|9600)/(3217|9600)	(18/32/28)/(26/43/25)	(60.5/34.0/59.1)/(36.3/22.0/39.5)	(14312/39268)/(13535/38981)	4.1/4.1	70.0/20.0	1.6/2.6
28	expression_database	765/981	43/43	(93.0|97.0)/(95.0|98.0)	94.4/95.6	25/27	(2519|7200)/(2468|7200)	(6/12/14)/(7/12/16)	(15.6/7.8/3.5)/(12.6/7.4/3.3)	(7866/21786)/(6870/19776)	6.2/5.0	40.0/40.0	16.4/26.8
29	Matchmaker	2761/3281	16/16	(92.0|95.0)/(95.0|98.0)	99.4/99.5	48/52	(4959|14400)/(4827|14400)	(20/31/18)/(26/52/19)	(1.2/0.8/1.3)/(0.9/0.5/1.3)	(11973/34155)/(10645/31521)	7.2/6.6	80.0/90.0	39.1/17.3
30	libtiff(tiffcp) v4.0.10	3536/2448	59/59	(94.0|99.0)/(95.0|98.0)	98.3/97.6	9/10	(1794|5200)/(1754|5200)	(6/8/17)/(5/7/18)	(140.7/105.5/N/A)/(55.2/39.4/N/A)	(9155/25845)/(8016/22312)	5.6/4.9	70.0/50.0	5.2/9.5
31	jhead v3.03	511/320	63/63	(91.0|94.0)/(93.0|97.0)	87.7/80.3	6/5	(1034|3000)/(1030|3000)	(2/2/12)/(2/2/11)	(49.8/49.8/1.2)/(25.1/25.1/2.5)	(5454/15665)/(4986/14651)	6.1/5.8	70.0/60.0	9.2/9.2
32	xpdf(pdftopng) v4.01.01	33317/32590	497/497	(94.0|96.0)/(95.0|98.0)	98.5/98.5	35/34	(223506|64000)/(233086|67877)	(4/8/32)/(5/9/34)	(348.3/174.1/N/A)/(693.2/385.1/N/A)	(50113/144634)/(49465/141123)	4.0/2.9	90.0/90.0	10.7/6.2
33	lodepng v20190928	728/679	102/102	(95.0|99.0)/(95.0|97.0)	86.0/85.0	7/8	(1786|5200)/(1856|5200)	(15/21/14)/(12/18/17)	(7.9/5.7/N/A)/(9.1/6.1/N/A)	(9498/26131)/(9375/25612)	5.0/4.9	70.0/80.0	12.3/3.5
34	crashmail v1.6	776/589	216/216	(94.0|98.0)/(93.0|96.0)	72.2/63.3	4/5	(1204|3600)/(1207|3600)	(8/15/12)/(4/7/13)	(70.0/37.3/46.7)/(93.3/53.3/28.7)	(2253/6172)/(1836/5412)	1.7/1.5	70.0/60.0	4.8/3.6
35	Netperf v2.6.0	17982/12764	8220/8220	(95.0|97.0)/(92.0|95.0)	54.3/35.6	4/3	(1307|3600)/(1215|3600)	(3/4/10)/(3/4/11)	(3254.0/2440.5/976.2)/(1514.7/1136.0/413.1)	(1648/4761)/(1807/5412)	1.3/1.5	50.0/40.0	17.0/20.0
36	pdfresurrect v0.15	23455/20597	1023/1023	(95.0|99.0)/(94.0|98.0)	95.6/95.0	24/24	(8580|24000)/(8598|24000)	(12/13/27)/(10/18/23)	(342.7/316.4/N/A)/(513.6/285.3/N/A)	(18993/54312)/(23467/65412)	3.0/2.9	80.0/80.0	27.0/27.3
37	sipp v3.3	4098/3981	2208/2208	(95.0|97.0)/(91.0|94.0)	46.1/44.5	6/5	(1238|3600)/(1254|3600)	(3/7/9)/(2/3/7)	(630.0/270.0/210.0)/(886.5/591.0/253.3)	(1744/5121)/(1383/4123)	1.4/1.1	50.0/40.0	7.5/7.8
38	sc v7.16	2381/2098	1029/1029	(93.0|96.0)/(94.0|99.0)	56.8/51.0	6/5	(1250|3600)/(1286|3600)	(4/6/12)/(4/6/11)	(338.0/225.3/112.7)/(267.3/178.2/97.2)	(1661/4781)/(1662/4651)	1.3/1.3	60.0/50.0	4.8/4.4
39	jasper v2.0.14	5762/4761	764/764	(95.0|99.0)/(96.0|98.0)	86.7/84.0	8/9	(3268|9600)/(3353|9600)	(4/7/13)/(5/7/12)	(397.2/227.0/89.9)/(346.5/247.5/19.4)	(7242/20182)/(7155/19871)	3.0/2.2	40.0/10.0	3.9/4.7
40	listswf(libming v0.4.7)	3478/2668	33/33	(93.0|96.0)/(94.0|99.0)	99.1/98.8	8/7	(2432|7200)/(2511|7200)	(5/7/14)/(4/6/15)	(44.3/31.6/N/A)/(85.8/57.2/N/A)	(6958/19888)/(6518/17872)	3.4/3.0	30.0/20.0	2.2/2.3
41	GraphicsMagick v1.3.26	5178/4921	68/68	(93.0|97.0)/(95.0|98.0)	98.7/98.6	8/9	(3299|9600)/(3388|9600)	(5/7/21)/(4/6/19)	(284.8/203.4/N/A)/(519.3/346.2/N/A)	(7286/21731)/(7919/22123)	2.5/2.5	40.0/20.0	1.6/1.5

num_1/num_2: for case No.n, num_1 is data value of crashing input 1; num_2 is data value of crashing input 2;
($percent_1$|$percent_2$): for the fitting rate of case No.n, $percent_1$ is average rate; $percent_2$ is maximum rate;
CGC program: case No.1 to case No.29;
Real-world program: case No.30 to case No.41;