DiffyFuzz: Input Generation with Differentiable Functions

FABRICE HAREL-CANADA, UCLA
APOORV GARG, UCLA
AISHWARYA DEV, UCLA

Well established whitebox testing techniques like symbolic and concolic execution cannot solve many real-world constraints due to limitations in their SAT solvers. In this work, we introduce a novel test generation framework called DiffyFuzz, which automatically approximates arbitrary program logic via a differentiable neural network and then "adversarially attacks" it to generate any target input. Our approach can be used as a standalone framework, but can also extend existing techniques to leverage the the benefits of both. In this work, we extend a symbolic fuzzing tool to cover uncovered branches where the condition is too complex to be solved by SAT solvers such as Z3. We demonstrate that our approach can expand branch coverage beyond the baseline tool with a small set of efficiently crafted inputs.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; • **Computing methodologies → Machine learning**.

Additional Key Words and Phrases: fuzzing, testing, datasets, neural networks, optimization, adversarial attacks

ACM Reference Format:
Fabrice Harel-Canada, Apoorv Garg, and Aishwarya Dev. 2022. DiffyFuzz: Input Generation with Differentiable Functions. 1, 1 (March 2022), 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Well established testing techniques like symbolic [13] and concolic [10] execution analyze a program to determine what inputs cause different code paths to execute. However, such approaches are limited by the capabilities of their underlying SAT solvers and many branch conditions cannot be (efficiently) resolved in practice [5]. This raises the question of how to access uncovered code when the branch conditions are sufficiently complex or otherwise out of reach of current testing techniques? Also, with rise in complexity, the time for solving these constraints are unknown as these SAT solvers are treated as black-box. So the output of running these type of executions under some time constraint can prove to be highly variable.

Our general approach to reaching uncovered code guarded by complex branch conditions is to approximate application logic using a differentiable function that can be efficiently manipulated via gradient-based techniques in convex optimization [2]. More specifically, we plan to leverage neural networks as universal function approximators [9] to mimic the behavior of small blocks of code responsible for setting variables guarding uncovered branches. Once trained, this differentiable surrogate function will be "attacked" via projected gradient descent (PGD) [14] to generate inputs that can reach previously unexecuted code.

Authors' addresses: Fabrice Harel-Canada, fabricehc@cs.ucla.edu, UCLA; Apoorv Garg, apoorv.garg.cse15@gmail.com, UCLA; Aishwarya Dev, aishwaryadev30@g.ucla.edu, UCLA.

1.1 Motivating Example

```
1 def neuzz_fn(a, b):
2     z = math.pow(3, a + b)
3     if z < 1:
4         return 0
5     elif z < 2:
6         return 1
7     elif z < 4:
8         # vulnerability
9         return 2
10    else:
11        return 3
```

Fig. 1. Modified version of Figure 3. from Neuzz [17]

As a motivating example, consider the following code sourced from the Neuzz paper [17] (Figure 1). This simple Python snippet demonstrates a general switch-like code pattern commonly found in many real-world programs. In particular, the example code computes a non-linear exponential function of the input and returns different values based on the output range of the `math.pow` function. The key challenge here is to find values of `a` and `b` that will trigger all possible branches, especially the branch on line 7 that guards potential vulnerabilities. Greybox testing frameworks like AFL [20] will not leverage any of the program logic in their (evolutionary) search for suitable test inputs. Whitebox techniques like symbolic and concolic execution will attempt to use statically or dynamically sourced program information, but due to reliance on limited SAT solvers, will not be able to (efficiently) handle the exponential function in line 2.

In contrast, our differentiable approximation and gradient-guided mutations are designed to exploit the function surface shape and quickly arrive at suitable inputs to access any branch in the function. This particular function can be modeled as a classification problem and we will show that a simple, 2-layer neural network can attain 100% accuracy on it after training for less than 2 seconds. We then use the neural approximator to perform an effective, gradient-guided optimization to incrementally refine values for `a` and `b` until each desired branch condition is solved, unlocking previously unexecuted and potentially vulnerable code.

2 RELATED WORKS

2.1 Neural Network Supported Fuzzing

Neuzz [17] is a recent fuzz testing technique that utilized neural networks in their algorithm. However, they used the networks to model *program branching behavior* while we intend to *approximate program functionality directly*. Both approaches are challenging in their own respects, but modeling arbitrary numerical programs involves significantly more engineering overhead to accomodate many different types of input-output mapping functions (e.g. regression vs. classification).

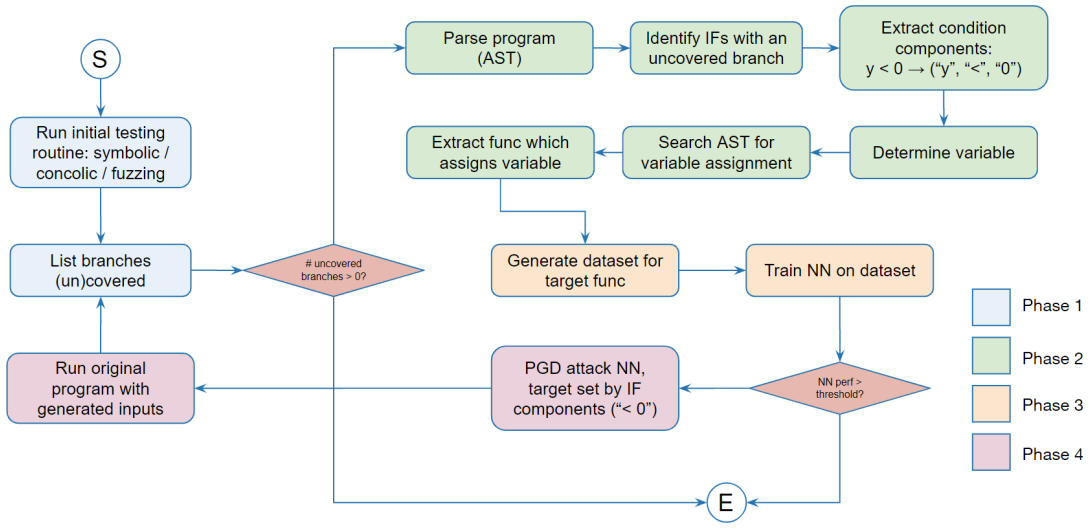


Fig. 2. Overview of DiffyFuzz

ATTuzz [21] is another approach that improves upon Neuzz in not only identifying the relevant bytes but also selecting mutation operators to cover uncovered branches. This approach focuses on dynamic and adaptive identification of most rewarding test inputs as seeds during Fuzzing with the help of deep learning models with attention mechanisms learned from fuzzing data to identify effective mutations on specific bytes of identified seeds. The focus of this approach is also exclusively on improved coverage while in addition to addressing the coverage, we intend to neurally simulate numerical program behavior.

2.2 Neural Programming

The related research area of neural programming [11] essentially uses neural networks to learn a latent representation of the target program's logic. Our work explores this idea in the more constrained application to small code blocks rather than computer memory management and generalized differential computing.

3 METHODOLOGY

We describe the different phases of our DiffyFuzz approach in detail below. Also see Figure 2 for a high-level visualization of the workflow.

3.1 Initializing a Coverage Profile

In the first phase, we create a *coverage profile* of the program under test (PUT) using an existent testing tool — e.g. a (symbolic) fuzzer. The tool attempts to generate up to 100 test inputs, which are then used to profile program coverage. The primary output of this phase is a list of covered and uncovered branches to be used downstream in identifying blocking code. Each branch is paired with a Boolean indicating whether the `True` or `False` path was taken or not. Since our tool is implemented in Python, coverage is provided using the self-reflection capabilities built into the language.

While it is possible to use DiffyFuzz as a standalone testing technique, our default implementation extends an existing tool. Prior theoretical work by Böhme and Paul [4] has highlighted the time-efficiency trade-off between random and systematic testing techniques. Since our approach is highly systematic and targeted, we recommend using it only after a brief round of random test input generation and allow our approach to access the more challenging branches. In this respect, our implementation combines both random and systematic testing and realizes the efficiency benefits both. Of course it is also possible to extend other systematic testing techniques, such as symbolic and concolic execution, as well.

3.2 Identifying Blocking Code

In the second phase, we identify *blocking code*, which we define as the set of lines responsible for setting a variable used to guard an uncovered branch of the program.

If the set of uncovered branches from the coverage profile is not empty, we begin by parsing the program into an AST and extract the nodes corresponding to the uncovered branches by matching up `lineno` attributes. Then we parse the condition node to extract the left operand, right operand, and comparison operation (e.g. <, >, ==). Ideally, one of the operands is identified to be a variable and the other a constant. If both of the operands are variables, we attempt to resolve one as constant by traversing the AST and determining the point of assignment. If assignment is made via some other code block or function call earlier in the program, we extract this code and it becomes the target of our next phase - function approximation.

The two primary outputs for this phase are: 1) a function (in memory) used to model a differentiable approximation (Section 3.3 and 2) a list of associated operand-target value pairs for guiding input generation (Section 3.4).

3.3 Function Approximation

In the third phase, we approximate a subset of program behavior directly. This entails two substeps: 1) dataset generation and 2) training the approximator.

3.3.1 Dataset Generation. Our approximator function requires a training set in order to learn the targeted program behavior. In order to create this dataset, we inspect the function `argspecs` to determine the number of inputs and outputs the function features. The number of inputs is determined from the number of arguments to the function and the number of outputs is determined via the number of `return` statements. If the number of outputs is greater than 1, we assume we are in the classification setting, else in the regression setting.

Then we use `pyfuzz` [7] to generate 1000 random float values per argument of the function. If the function accepts multiple arguments, we apply a cross product to acquire all possible combinations of values and then sample 1000 of them to maintain a reasonable dataset size. These fuzzed data points are then passed into the extracted function to generate the ground truth targets – i.e. labels in the classification setting. Both the inputs and the ground truth outputs can be optionally normalized. Normalization is a critical preprocessing step that improves generalization and expedites convergence [1, 18]. In particular, min-max scaling has been found to empirically improve training more than competing alternatives [1], so we implemented one in PyTorch. All values are scaled to be between 0 and 1. Finally, we split the dataset into train and test sets at a 90-10 ratio.

3.3.2 Training an Approximator. Now that we have a model and a dataset, a differentiable function must be trained. While there are a handful of options available to us, we elected to use shallow 2-layer neural network because of its well known connection with universal approximation theory [9]. The network is implemented in PyTorch [16], uses linear layers with a dimension of 512 units, and LeakyReLU activations [19]. We use `pytorch-lightning` [8] to

support the training and testing pipelines. Even without a GPU, most of our studied functions can be well approximated in 1-3 seconds when trained on 900 data points for 3 epochs. The final step of the training process is to extract the input and output scalers from the dataset generator to create a prediction pipeline. This allows the approximator to accept inputs in their human-interpretable ranges, scale them for model processing, and then inverse scale the model outputs back to the expected output range. For example, if we were to input the value 212.0 to an approximator of the `fahrenheit_to_celcius_disc_fn` extracted from one of the subject programs, it would first be scaled to 0.9157, passed into the model to output 0.9160, which is then inverse scaled to 99.9513 — very close to the expected output of 100.0. Note again that the inputs and outputs are scaled independently so despite their proximity in scaled space, they correspond to very different values in unscaled space.

The competing objectives in this step are to simultaneously maximize model performance while minimizing training time. The number of layers in the model, size of the dataset, and number of training epochs were all empirically set based on small-scale studies. Appendix A shows one such experiment for setting the default number of epochs based on the loss logs collected during training. Other studies are omitted due to page space limitations.

Before progressing to the final phase of input generation, we rate the performance of the model on a test set. Only models with a mean loss ≤ 0.1 or an accuracy ≥ 0.8 will be used and all others will exit the DiffyFuzz algorithm. In the future, we plan to add a more sophisticated model search routine that will explore increases to approximator type (i.e. another ype of universal approximator), model depth / width, training set size, and training time until one of the aforementioned thresholds are satisfied.

3.4 Generating Targeted Inputs

In this phase, we have now obtained a sufficiently well-trained and differentiable approximator that can smoothly generalize the observed program behaviors — modeling potentially non-linear and non-convex behaviors. The approximator can then be used for efficiently computing gradients and higher-level derivatives to guide the fuzzing input generation process.

Different gradient-guided optimization techniques like gradient descent, Newton's method, or quasi-Newton methods like L-BFGS can use gradient or higher-order derivatives for faster convergence [2, 3, 15]. Our approximators can potentially use any of these techniques, although we have opted to leverage recent work in adversarial attacks from the machine learning literature because of their ability to support more flexible optimization objectives [6].

The specific technique we used for our approach is projected gradient descent (PGD), an iterative, bounded, and targeted whitebox attack that is regarded as a *universal* "first-order adversary", i.e. the strongest attack utilizing first order information about the network [14]. It is bounded in the sense that perturbations injected into the seed inputs are controlled to be of a certain magnitude and targeted in the sense that we specify the output we'd like to see from the approximator. This in turn allows us to generate inputs that will also trigger similar outputs in the original program functionality and therefore allows our approach to access any branch guarded by its logic.

We also extended the standard PGD implementation to make it quicker, more targeted, and applicable to the regression task setting. During each iteration of the PGD process, the loss is used to derive the gradients w.r.t. the seed input and a "step" is taken in the direction which minimizes the distance to the target. However, it is possible to overstep the target and get stuck oscillating between two suboptimal points. We solve this by implementing an *interval halving* method to dynamically set the allowable perturbation per iteration with an exit condition when the perturbation size is smaller than $1e-6$. This has the dual benefit of terminating the algorithm earlier than the maximum allowable steps and arriving at the target with a greater degree of precision. In general, this permits tests to be generated in well under

a second each. Secondly, by default, nearly all adversarial attacks are oriented towards the classification setting and use a hard-coded cross-entropy loss function. We generalized several parts of the algorithm to accept an arbitrary loss function and use the L_1 loss for the regression setting.

Our custom input generator requires 3 main arguments — an approximator, an operand, and a target. The approximator was sourced from Section 3.3 and was heuristically judged to be of sufficient performance. The operand and targets are sourced from Section 3.2. If the operand is an equivalence comparison (i.e. ==) then the target is passed directly to the PGD-based generator. If the operand is a greater than (or equal to) comparison (i.e. $>, \geq$) then we transform the target by adding 1 (to avoid complications where the original target is 0) and multiplying by 255. This sets a new internal objective that is much larger than the original target and yields some value which satisfies our external objective. If the operand is a less than (or equal to) comparison (i.e. $<, \leq$), we perform a similar transformation as above, only negated. For example, if the given operand and target are "<" and "0", respectively, our generator will set an objective of -255 and attempt to generate a corresponding input. This input can then be passed as a test to the program to access the associated branch and potentially discover defective program behavior.

4 EVALUATION

In this section we evaluate our approach by applying the technique to several case studies. In particular, we ask the following research questions:

RQ1. How well can arbitrary program logic by approximated by a neural network?
RQ2. How well and how quickly can DiffyFuzz increase branch coverage beyond existent baseline techniques?

4.1 Evaluation Setup

4.1.1 Research Protocol. In our evaluation, we generate test suites and measure the branch coverage using Python's internal reflection capabilities via `sys.settrace`. In general, a branch may consist of line code pairs, but in our case we only count those associated with `if`, `for`, and `while` AST nodes for the sake of clarity.

For each testing technique, we generate at most 1000 inputs and capture the following performance: branch coverage (%), cumulative branch coverage by number of inputs generated, number of inputs generated in 60 seconds, and the number of exceptions raised.

4.1.2 Subject Programs. Our case studies consist of 14 toy subject programs representing three different classes of functions:

(1) **Continuous:** *n* `float` inputs, 1 `float` output as the target in a regression task setting
(2) **Discontinuous:** *n* `float` inputs, *m* `int` outputs as classification labels
(3) **Compositions:** *n* `float` inputs, *m* `int` outputs, where multiple functions are chained together

Each program contains a number of branching conditions that guard additional program logic and deliberately injected, but hard-to-reach exceptions. The primary goal is to trigger such exceptions using the structural coverage metric of branch coverage to guide test exploration and generation. Note that only functions that can be represented numerically as input-output mappings are in scope for DiffyFuzz.

4.2 RQ1. How well can arbitrary program logic by approximated by a neural network?

In order to determine whether DiffyFuzz can be efficiently used to expand coverage of hard-to-reach branches, it must first be capable of approximating the program logic guarding said branches. For each of our subject programs, we

extracted the functions responsible for setting the blocking variable and trained 2-layer neural networks using all of the configuration details described in Section 3.3.

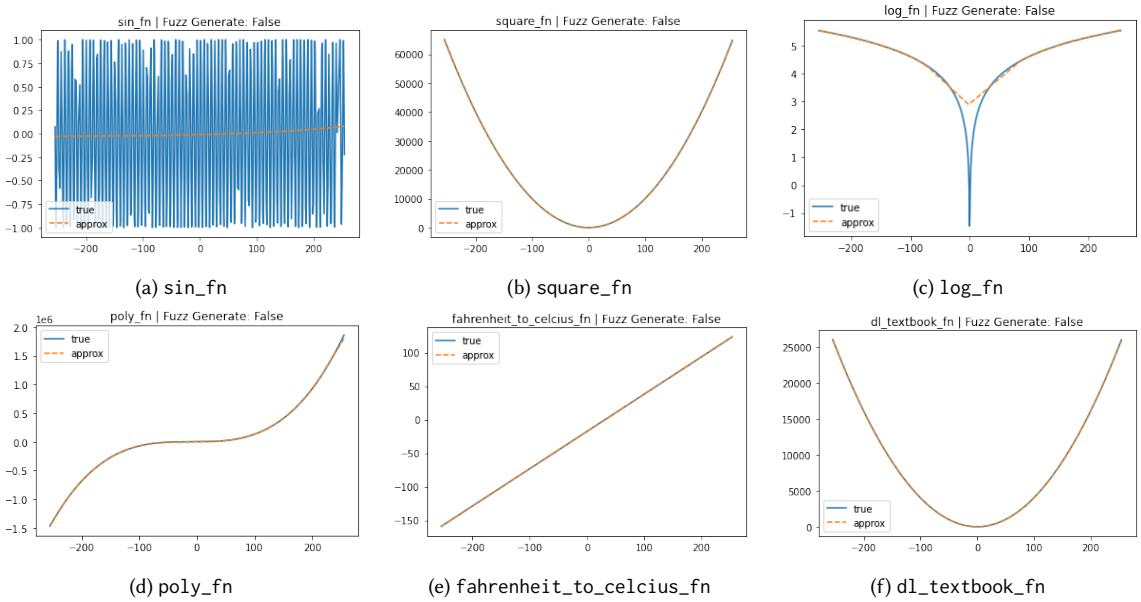
We automatically infer that the task setting is either regression or classification based on the number of outputs in the original function. This in turn effects the model architecture, loss function, and performance metrics reported for the model performance. For all settings, the test loss is reported as the primary indicator of quality and for the classification setting we also report accuracy on the held-out test set.

Table 1 shows the results of training with 900 inputs for 3 epochs with early stopping. In general, all three types of functions are very well approximated using our approach. For example, 6 out of 7 functions had 100% accuracy on the test set and the remaining one still scored highly at around 91%. On the continuous side, the test losses are all relatively low with the exception of the `sin_fn`, which does not appear to be an easy target for neural approximation, though it is unclear why that may be from the loss.

Subject Program	Function Name	Type	Test Loss	Test Accuracy	Training Time (s)
program_1.py	sin_fn	continous	0.34587	NA	1.60
program_2.py	square_fn	continous	0.00689	NA	1.57
program_3.py	log_fn	continous	0.03056	NA	1.58
program_4.py	poly_fn	continous	0.01067	NA	1.62
program_5.py	pythagorean_fn	continous	0.01103	NA	1.61
program_6.py	fahrenheit_to_celcius_fn	continous	0.00041	NA	1.61
program_7.py	dl_textbook_fn	continous	0.00299	NA	1.61
program_8.py	square_disc_fn	discontinous	1.36842e-06	1.0	1.62
program_9.py	log_disc_fn	discontinous	0.00073	1.0	1.63
program_10.py	neuzz_fn	discontinous	1.28657e-05	1.0	1.61
program_11.py	fahrenheit_to_celcius_disc_fn	discontinous	0.26717	0.91	1.60
program_12.py	log_sin_fn	composition	2.69615e-06	1.0	1.60
program_13.py	f_of_g_fn	composition	1.43994e-06	1.0	1.66
program_14.py	arcsin_sin_fn	composition	7.15119e-06	1.0	1.62

Table 1. Approximator performance on target functions extracted from our subject programs. For functions with continuous outputs (i.e. regression models), a low test loss indicates higher performance. For functions with discontinuous outputs (i.e. classification models), we additionally provide an accuracy metric. In most cases, the program logic is very well modeled by our swiftly trained approximators.

While the test loss is a popular *quantitative* metric for model performance, it is not an especially informative one. To supplement it with a more *qualitative* notion of quality, we also plotted the model predictions and ground truths against their inputs to visualize the degree of overlap. The results in Figure 3 provides some additional insights. The first is that approximation can be successful for only specific ranges of the input space as seen with the `log_fn` in Subfigure 3c — a fact obscured by a single number score. Such approximators may still provide enough guidance for gradient optimization to solve the more open-ended inequality constraints, but are unlikely to be helpful with the more strict equalities. The second is that it functions with steep, sharp, or highly oscillatory output surfaces are especially challenging to neurally approximate. In the future we may want to incorporate fallback approximators such as a taylor polynomial. Fortunately, the remaining functions show a significant degree of overlap between the ground truth and approximations, further supporting the likelihood that DiffyFuzz will be able to assist with highly efficient, gradient-guided input generation.



(a) `sin_fn` (b) `square_fn` (c) `log_fn`

(d) `poly_fn` (e) `fahrenheit_to_celcius_fn` (f) `dl_textbook_fn`

Fig. 3. Comparison between ground truth function and neural approximations. Note that only functions with 1 input and 1 output are visualized.

4.3 RQ2. How well and how quickly can DiffyFuzz increase branch coverage beyond existent baseline techniques?

Out of 15 subject programs, there were 8 subject programs where we could not perform symbolic execution as they were using functions from libraries such as numpy, scipy. The functions specified in branch conditions were written in source code of the library which was in native C and thus uncovered by symbolic execution. For those programs, we have marked the branch coverage as 0%.

Subject Program	Function Name	Sym Coverage	Sym Time (s)	DF Coverage	DF Time (s)
program_1.py	sin_fn	0.0%	0	66.67%	1.07
program_2.py	square_fn	83.33%	5.99	83.33%	1.11
program_3.py	log_fn	0.0%	0	33.33%	1.36
program_4.py	poly_fn	83.33%	12.46	83.33%	1.05
program_5.py	pythagorean_fn	0.0%	0	0.0%	1.54
program_6.py	fahrenheit_to_celcius_fn	100.0%	3.25	NA	NA
program_7.py	dl_textbook_fn	0.0%	0	33.33%	1.16
program_8.py	sin_disc_fn	0.0%	0	100.0%	1.68
program_9.py	square_disc_fn	75.0%	647.40	75.0%	1.40
program_10.py	log_disc_fn	0.0%	0	100.0%	1.61
program_11.py	neuzz_fn	75.0%	195.81	87.5%	1.25
program_12.py	fahrenheit_to_celcius_disc_fn	83.33%	28.99	83.33%	1.29
program_13.py	log_sin_fn	0.0%	0	50.0%	1.59
program_14.py	f_of_g_fn	50.0%	6.38	50.0%	1.68
program_15.py	arcsin_sin_fn	0.0%	0	50.0%	1.61

Table 2. Coverage obtained by Symbolic Execution and Diffy Function Apporach along with the execution time. Execution time also includes the time to train the model. In most cases, we can see a definite increase in branch coverage when applying differntiable function to cover complex branch conditions.

For 7 programs, we can see some amount of improvement. For example, for subject program9, despite taking more than 3 minutes, symbolic execution was stuck at 75%, while Diffy function took merely 1.25 seconds to gain 12.5% which is considerable. Though, there are functions which do not show significant rise in branch coverage, we plan on discovering the root cause and analyze it to provide feedback to better our model.

5 DISCUSSION AND FUTURE WORK

In this section, we discuss potential threats to validity interweaved with potential future work to address them.

5.1 Threats to Validity

Internal Validity. The main threat to internal validity is the suitability of our neural networks to approximate arbitrary program behavior. On a deeper level, this threat applies to any approach involving a learned function because all of them must be trained on a specific distribution of the input space. The immediate neighborhood around the training points may be well internalized, but performance degrades steeply in areas the model has not previously seen. Since the space of possible real inputs is infinite, there will always be areas for which the learned approximator fails to generalize.

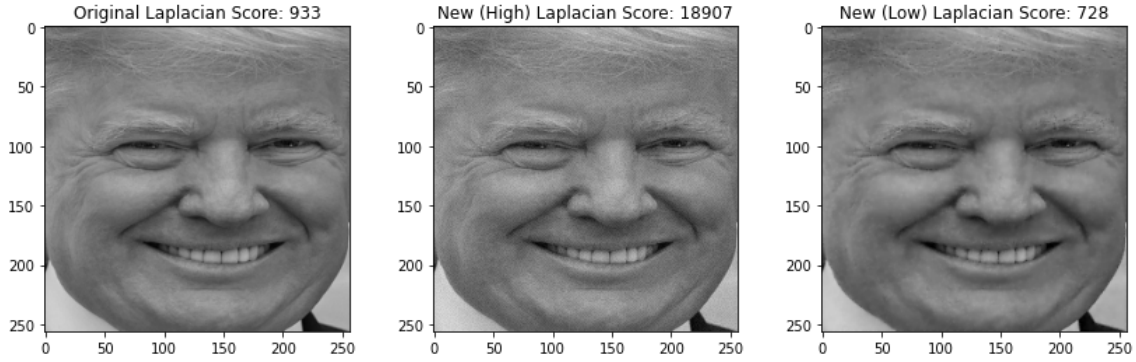
Another issue connected to the particular training distribution is that of class imbalance. Learned functions typically perform poorly when some classes are underrepresented or not present at all in the training set [12]. Due to the need to generate an initial training data from scratch via fuzzing, we are not able to guarantee an even distribution of classes or even the presence of a class at all. In the future, this issue may be mitigated by generating more data from a wider possible range and subsampling to artificially induce class balance at the cost of increased execution time for DiffyFuzz.

External Validity. Threats to external validity relate to the generalizability of the experimental results. In our case, this is specifically related to the subject programs used in the experiments. We acknowledge that we have only experimented with a limited set of toy programs. However, much of the underlying structure of these programs were inspired by real-world code[1]. Furthermore, this work represents an entirely novel approach to test input generation

[1]https://github.com/syaringan357/Android-MobileFaceNet-MTCNN-FaceAntiSpoofing

and we assert that the subject programs cover enough functionality to constitute a solid first step in this new area with no citable prior research.

Future research would broaden the subject programs to include publicly available code containing more complex functions like image arrays. In an initial experiment along this thread, we attempted to approximate a real-world `lap_score` function from the aforementioned Android app which convolves a laplacian kernel over an image to estimate the likelihood that it has been digitally manipulated. Our preliminary results show that it is possible to manipulate an input image using our gradient-guided approach and successfully trick the function to access both sides of a subsequent branch in the program logic (e.g. `if lap_score(img) > 1000`).



(a) Original Laplacian Score (b) Heightened Laplacian Score (c) Lowered Laplacian Score

Fig. 4. Manipulating the laplacian score of an image via gradient-guided mutations. The subsequent images can be used to access both sides of the condition (e.g. `if lap_score(img) > 1000`) without deviating significantly in human-perceptibility.

6 CONCLUSION

In this paper we have presented a novel testing approach that allows engineers to access code guarded by complex conditions. Current whitebox techniques are limited by the capabilities of their underlying SAT solvers while our approach works with most kinds of arbitrary program logic. By training a model to mimic targeted blocks of program logic, we learn a differentiable function that can be paired with gradient-guided mutations to efficiently generate inputs that target all uncovered branches. Our results show that branch coverage can be substantially improved over a purely symbolic baseline without dramatically increasing the generation time.

REFERENCES

[1] Gokhan Aksu, Cem Güzeller, and Taha Eser. 2019. The Effect of the Normalization Method Used in Different Sample Sizes on the Success of Artificial Neural Network Model. *International Journal of Assessment Tools in Education* 6 (07 2019), 170–192. https://doi.org/10.21449/ijate.479404

[2] Stephen Boyd and Lieven Vandenberghe. 2004. Convex optimization.

[3] Richard H. Byrd, Peihuang Lu, Jorge Nocedal, and Ciyou Zhu. 1995. A limited memory algorithm for bound constrained optimization. *SIAM Journal of Scientific Computing* 16 (Sept. 1995), 1190–1208. https://doi.org/10.1137/0916069

[4] Marcel Böhme and Soumya Paul. 2016. A Probabilistic Analysis of the Efficiency of Automated Software Testing. *IEEE Transactions on Software Engineering* 42, 4 (2016), 345–360. https://doi.org/10.1109/TSE.2015.2487274

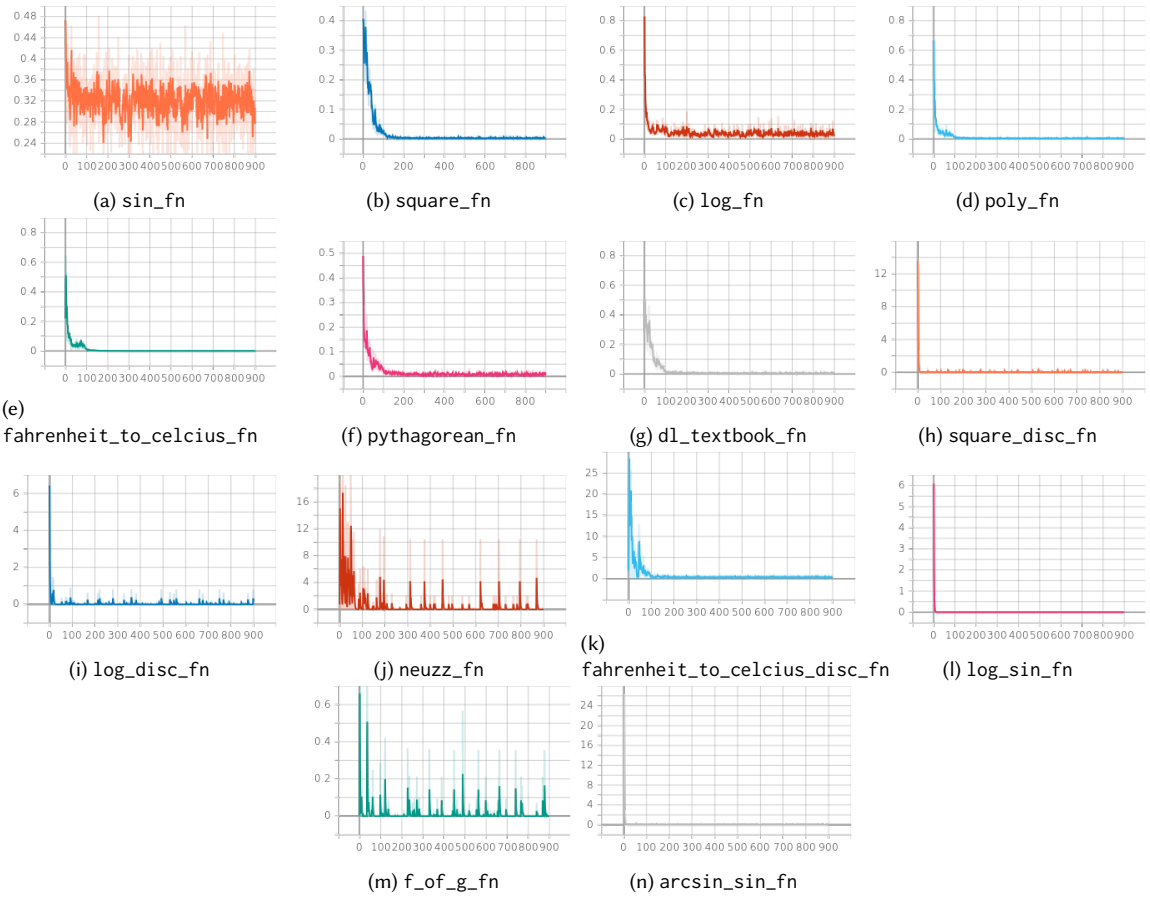
[5] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (feb 2013), 82–90. https://doi.org/10.1145/2408776.2408795

[6] Yinpeng Dong, Fangzhou Liao, Tianyu Pang, Xiaolin Hu, and Jun Zhu. 2017. Discovering Adversarial Examples with Momentum. arXiv:1710.06081 http://arxiv.org/abs/1710.06081

[7] Fabrice Harel-Canada et al. 2021. Pyfuzz. https://github.com/fabriceyhc/pyfuzz.
[8] William Falcon and The PyTorch Lightning team. 2019. PyTorch Lightning. https://doi.org/10.5281/zenodo.3828935
[9] Ken-Ichi Funahashi. 1989. On the approximate realization of continuous mappings by neural networks. *Neural Networks* 2, 3 (1989), 183–192. https://doi.org/10.1016/0893-6080(89)90003-8
[10] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 213–223. https://doi.org/10.1145/1065010.1065036
[11] Alex Graves, Greg Wayne, and Ivo Danihelka. 2014. Neural Turing Machines. arXiv:1410.5401 http://arxiv.org/abs/1410.5401
[12] Justin Johnson and Taghi Khoshgoftaar. 2019. Survey on deep learning with class imbalance. *Journal of Big Data* 6 (03 2019), 27. https://doi.org/10.1186/s40537-019-0192-5
[13] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (jul 1976), 385–394. https://doi.org/10.1145/360248.360252
[14] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2019. Towards Deep Learning Models Resistant to Adversarial Attacks. arXiv:1706.06083 [stat.ML]
[15] Jorge Nocedal. 1980. Updating quasi-Newton matrices with limited storage. *Mathematics of computation* 35, 151 (1980), 773–782. https://doi.org/10.1090/S0025-5718-1980-0572855-7
[16] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. , 8024–8035 pages. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf
[17] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2019. NEUZZ: Efficient Fuzzing with Neural Program Smoothing. arXiv:1807.05620 [cs.CR]
[18] J. Sola and J. Sevilla. 1997. Importance of input data normalization for the application of neural networks to complex industrial problems. *IEEE Transactions on Nuclear Science* 44, 3 (1997), 1464–1468. https://doi.org/10.1109/23.589532
[19] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. 2015. Empirical Evaluation of Rectified Activations in Convolutional Network. arXiv:1505.00853 http://arxiv.org/abs/1505.00853
[20] Michal Zalewski. 2013. American fuzzy lop (2.52b). https://lcamtuf.coredump.cx/afl/
[21] Shunkai Zhu, Jingyi Wang, Jun Sun, Jie Yang, Xingwei Lin, Liyi Zhang, and Peng Cheng. 2021. Better Pay Attention Whilst Fuzzing.

A TRAINING SETTINGS

In order to determine the training time for our approximator networks, we plotted the training loss and found that most functions converged around 200 steps (a little over 2 epochs). We therefore chose a default number of training epochs to be 3 and implemented early stopping if the training loss could not be improved after 3 steps. See Figure 5 for visualization of the training losses by batch / step.



(a) `sin_fn` (b) `square_fn` (c) `log_fn` (d) `poly_fn`

(e) `fahrenheit_to_celcius_fn` (f) `pythagorean_fn` (g) `dl_textbook_fn` (h) `square_disc_fn`

(i) `log_disc_fn` (j) `neuzz_fn` (k) `fahrenheit_to_celcius_disc_fn` (l) `log_sin_fn`

(m) `f_of_g_fn` (n) `arcsin_sin_fn`

Fig. 5. Training Loss by Step for various target functions extracted from subject programs.