Computer Standards & Interfaces 94 (2025) 104011

Contents lists available at ScienceDirect

Computer Standards & Interfaces

journal homepage: www.elsevier.com/locate/csi

Leveraging activation and optimisation layers as dynamic strategies in the multi-task fuzzing scheme

Sadegh Bamohabbat Chafjiri *, Phil Legg, Michail-Antisthenis Tsompanas, Jun Hong

Computer Science Research Centre, University of the West of England, Bristol, UK

ARTICLE INFO

Keywords:
Fuzzing
Neural network
LReLU
Nadam optimisation
Sensitivity analysis

ABSTRACT

Fuzzing is a common technique for identifying vulnerabilities in software. Recent approaches, like She et al.'s Multi-Task Fuzzing (MTFuzz), use neural networks to improve fuzzing efficiency. However, key elements like network architecture and hyperparameter tuning are still not well-explored. Factors like activation layers, optimisation function design, and vanishing gradient strategies can significantly impact fuzzing results by improving test case selection. This paper delves into these aspects to improve neural network-driven fuzz testing.

We focus on three key neural network parameters to improve fuzz testing: the Leaky Rectified Linear Unit (LReLU) activation, Nesterov-accelerated Adaptive Moment Estimation (Nadam) optimisation, and sensitivity analysis. LReLU adds non-linearity, aiding feature extraction, while Nadam helps to improve weight updates by considering both current and future gradient directions. Sensitivity analysis optimises layer selection for gradient calculation, enhancing fuzzing efficiency.

Based on these insights, we propose LMTFuzz, a novel fuzzing scheme optimised for these Machine Learning (ML) strategies. We explore the individual and combined effects of LReLU, Nadam, and sensitivity analysis, as well as their hybrid configurations, across six different software targets. Experimental results demonstrate that LReLU, individually or when paired with sensitivity analysis, significantly enhances fuzz testing performance. However, when combined with Nadam, LReLU shows improvement on some targets, though less pronounced than its combination with sensitivity analysis. This combination improves accuracy, reduces loss, and increases edge coverage, with improvements of up to 23.8%. Furthermore, it leads to a significant increase in unique bug detection, with some targets detecting up to 2.66 times more bugs than baseline methods.

1. Introduction

Fuzzing has emerged as a powerful automated software testing technique, playing a critical role in identifying vulnerabilities and enhancing software robustness. It systematically subjects programs to dynamically generated inputs, uncovering flaws that could otherwise remain undetected [1,2].

Initially conceived as random input generation, fuzzing has since evolved into a sophisticated approach [3–7], leveraging structured [8–10] and feedback-driven methodologies [11,12] to improve test case generation efficiency. More recently, the integration of ML techniques [13] has further transformed fuzzing by enabling adaptive input generation, enhancing test case prioritisation, and optimising bug detection strategies [14,15,15–20] through better bug classification and automated bug analysis [21–25]. It has also advanced data flow interpretation, program property prediction, and guided mutation strategies, addressing ambiguity in defect identification [26–29].

Despite these advancements, significant challenges remain in enhancing the efficacy of ML-based fuzzing. Key areas such as activation functions, optimisation strategies, and post-training sensitivity analysis have not been sufficiently explored in the context of fuzzing, limiting potential gains in performance and vulnerability detection. This paper introduces LMTFuzz, an enhanced ML-driven fuzzing framework, which addresses these challenges by integrating advanced activation functions, optimisation algorithms, and sensitivity analysis in MTFuzz [30].

Specifically, we replace the standard ReLU [31] with Leaky ReLU (LReLU) [31] to mitigate the "dying ReLU" problem, and adopt the Nadam [32,33] optimiser in place of Adam to improve training stability and convergence speed. Additionally, we incorporate post-training sensitivity analysis to prioritise fuzzing test cases based on their gradient magnitudes [34]. Through these innovations, we aim to improve fuzzing performance, particularly in seed selection, vulnerability discovery, and overall fuzzing efficiency.

* Corresponding author.
E-mail address: sadegh.bamohabbatchafjiri@uwe.ac.uk (S. Bamohabbat Chafjiri).

https://doi.org/10.1016/j.csi.2025.104011
Received 10 October 2024; Received in revised form 14 March 2025; Accepted 31 March 2025
Available online 12 April 2025

This research addresses the following questions:

- How does the use of LReLU activation layers affect fuzzing performance in comparison to the baseline ReLU function employed in MTFuzz?
- What are the comparative effects of the Nadam optimisation technique versus the baseline Adam optimiser on enhancing fuzzing performance?
- To what extent does post-training sensitivity analysis influence the overall effectiveness of fuzzing outcomes?

To find answers to the questions posed above, this paper contributes to the field in the following key ways:

- First, we investigate the role of activation functions by analysing LReLU as an alternative to MTFuzz's standard ReLU. LReLU's allowance for small negative gradients in inactive units could improve gradient flow, mitigate dead neurons, and enhance model expressiveness in fuzzing applications. Given the iterative nature of fuzzing, better gradient flow could lead to improved convergence rates and deeper code coverage.
- Second, we examine Nadam as an alternative optimiser to Adam, examining the effect of Nadam's combination of Nesterov momentum and adaptive moment estimation could improve training stability and convergence speed in fuzzing models. While Adam has been widely used, Nadam's potential benefits, especially in the context of fuzzing, remain underexplored.
- Finally, we incorporate post-training sensitivity analysis to quantify the impact of different network layers on test case generation efficiency. By ranking input features based on their gradient magnitudes, this analysis could provide insights into how ML models prioritise fuzzing paths, ultimately leading to a more effective test case selection strategy.

These enhancements are designed to optimise ML-driven fuzzing by refining seed selection, enhancing vulnerability detection, and improving overall fuzzing efficiency through the introduction of a novel activation layer and a new optimiser, along with the proposal of a new sensitivity analysis—an element that MTFuzz currently lacks. While MTFuzz outperforms fuzzers like Neuzz [35], Angora [36], FairFuzz [37], AFL [38], and AFLFast [39], it still has limitations, such as inefficient seed selection and underutilised ML-driven optimisations. LMTFuzz addresses these deficiencies by optimising ML-driven approaches, offering notable improvements. We evaluate LMTFuzz through experiments comparing it exclusively to MTFuzz, which, due to its demonstrated superiority, serves as a strong baseline. Our comparison focuses on code coverage, vulnerability detection, and operational efficiency, advancing the integration of ML techniques in fuzzing for more effective software testing.

The structure of the paper is as follows: Section 2 provides the necessary background and related work to contextualise our study. Section 3 outlines the adopted methodology and experimental framework. Section 4 provides a detailed overview of the experimental setup, including hyperparameter tuning, the testing environment, and introduces the fuzzing targets. Section 5 presents observations based on the collected results and provides an in-depth evaluation of these findings. Section 6 explores potential avenues for future research, while Section 7 concludes with a summary of key insights and contributions.

2. Related work

To address the challenges in software security [40,41] and opportunities highlighted in Section 1, it is essential to examine the existing body of work that integrates optimisation strategies, neural network architectures, and fuzzing methodologies. The literature provides a comprehensive foundation on the application of Stochastic Gradient Descent (SGD) and its variants in deep neural networks (DNNs) and reinforcement learning (RL), both of which have demonstrated significant potential in advancing fuzzing techniques and improving software security.

In the context of DNNs, SGD has been a fundamental approach for model optimisation, widely applied in fuzzing tasks. For example, Grieco et al. introduced VDiscover, a system that utilises SGD to combine static and dynamic features with ML techniques such as random oversampling to detect memory corruptions in operating systems [42].

Additionally, GANFuzz integrated Generative Adversarial Networks (GANs) with Long Short-Term Memory (LSTM) networks and the policy gradient method to assess industrial network protocols, employing SGD with dropout and L2 regularisation techniques [43].

In another application, DeepSmith [44] leveraged standard SGD over 50 epochs with a learning rate schedule that decayed by 5% per epoch, ensuring stable training. The system used forget gates and voting heuristics for differential testing during compiler validation.

Beyond classical optimisation, gradient-based techniques have emerged as effective tools in guiding fuzzing processes. For instance, GradFuzz [45] introduced gradient vector coverage as a novel coverage metric to guide fuzzers towards crash-inducing paths in DNN-based fuzzing, using gradients to prioritise testing efforts.

Another approach involved guiding gradients smoothly using nonlinear techniques such as sigmoid, Softmax, and tanh functions. NEUZZ [35] extended this concept by employing gradient-guided input generation and program smoothing techniques like Gaussian and sigmoid (for the output layer), improving fuzzers' ability to efficiently explore more program states.

Recent advances have also explored the integration of SGD within reinforcement learning (RL)-based fuzzing approaches. Patil and Kanade [46] adapted AFL's heuristics into a contextual bandit framework, framing the fuzzing process as a reinforcement learning problem. This allowed dynamic adjustments of fuzzing iterations using the policy gradient method with nonlinear functions (tanh and Softmax), improving test case generation efficiency based on real-time feedback. Similarly, AgentFuzz [47] combined gradient-based optimisation with a DRL framework to mutate seeds in a deep learning model, identifying potentially vulnerable areas more efficiently. This was achieved by using gradient-based adversarial attack methods to generate diverse test cases capable of inducing larger loss values, thereby revealing system failures faster. PreFuzz [48] further enhanced gradient guidance and mutation effectiveness by introducing an efficient edge selection mechanism, reducing computational overhead and improving fuzzing outcomes.

Despite the success of SGD in these applications, challenges such as sensitivity to learning rate selection and vulnerability to local minima remain. To address these issues, advanced variants of SGD, such as Adam, have been proposed. These optimisers introduce adaptive learning rates and momentum terms, which help prevent overshooting during optimisation and provide resilience against noisy gradients, making them well-suited for complex fuzzing tasks where the underlying search space is highly dynamic.

Nichols et al. employed GANs alongside AFL for seed file initialisation in their "Faster Fuzzing" framework, using the binary cross-entropy loss function and training the model with ReLU as the inner activation layer, tanh as the output activation layer, and the Adam optimiser [49].

In V-Fuzz, Li et al. utilised SGD and its variant, the Adam optimisation algorithm, for both training and pretraining tasks in the context of vulnerability-oriented prediction, leveraging attributed control flow graphs [50]. In this approach, the tanh and sigmoid functions were employed as nonlinear activation functions, while ReLU served as the rectified linear unit. These methods minimised the cross-entropy loss function, with Adam providing adaptive learning rates and momentum to improve convergence and performance during pretraining. Together, SGD and Adam ensured efficient iterative optimisation of the model parameters throughout the training process.

Jeon and Moon's "Dr. PathFinder" [51] integrated DRL techniques with fuzzing by employing an RL agent to evaluate branch states during concolic execution, using the Adam optimiser. This approach prioritised the exploration of "deep" execution paths over "shallow" ones, reducing unnecessary exploration and enhancing memory efficiency.

A notable example of such integration is MTFuzz, proposed by She et al. [30], which is recognised as one of the most recent and effective machine learning (ML)-based fuzzing schemes. It utilises a multi-task neural network model with sigmoid activation and the Adam optimiser to guide the fuzzing mutation process. MTFuzz employs "hard parameter sharing" across multiple tasks, such as edge coverage, approach-sensitive edge coverage, and context-sensitive edge coverage. The loss functions are tailored to each task to maximise the overall effectiveness of the fuzzing process.

Despite the widespread adoption of ReLU and Adam in these applications, the literature reveals a notable lack of exploration into alternative activation functions and optimisation techniques. For instance, the LReLU activation function, which mitigates the "dying ReLU" problem by allowing a small, non-zero gradient for negative inputs, has seen limited application in the fuzzing domain, despite its potential to enhance model performance. Similarly, while Adam is commonly employed due to its adaptive learning rate and momentum, its more advanced variant, Nadam – which incorporates Nesterov momentum – remains underutilised in fuzzing-related research. This gap highlights a valuable opportunity for further investigation into how LReLU and Nadam could improve fuzzing effectiveness, particularly in scenarios involving complex neural network architectures or dynamic search spaces. Moreover, the sensitivity analysis of gradient calculations across different layers remains an underexplored area in the existing literature. These gaps underscore the potential for advancing the state of the art, providing a strong foundation for the present study to contribute meaningful insights into activation functions and optimisation strategies that can enhance fuzzing techniques.

3. Methodology

This research adopts a structured methodology to assess the impact of activation functions, optimisation techniques, and sensitivity analysis on the performance of a Multi-Task Neural Network (MTNN) fuzzing scheme. Our approach focuses on three primary components, each corresponding to a specific research question. Moreover, it investigates the collective influence of these techniques on overall fuzzing performance, particularly concerning edge coverage within a defined time constraint.

3.1. Understanding the baseline model

MTFuzz framework utilises an MTNN integrated with a "hard parameter sharing" approach, employing three distinct methodologies:

- *"Edge Coverage"*
- *"Approach-sensitive Edge Coverage"*
- *"Context-sensitive Edge Coverage"*

Approach-sensitive edge coverage evaluates the proximity of unexplored edges using a numerical scale from 0 to 1, allowing for efficient modifications of input data to target these edges.

Context-sensitive edge coverage identifies the call context of explored edges, providing insights into the program's internal states and enhancing the differentiation of inputs activating the same edge from distinct contexts.

MTFuzz employs a *compact embedding* technique to capture critical input features while minimising bitmap size. This approach reduces edge explosion and enhances performance via transfer learning, enabling effective sharing of representations across parsers for XML and ELF binaries.

Among these methodologies, both edge coverage and context-sensitive edge coverage are framed as classification tasks, whereas approach-sensitive edge coverage is structured as a regression task. Consequently, the loss functions for each methodology are tailored to their specific task requirements.

The architecture of the MTNN includes shared initial layers and task-specific output layers, allowing for a unified feature representation. Its overall loss function is a weighted combination of task-specific losses.

Algorithm 1 Logical Inconsistency in MTFuzz "if" statement for Rare Edge Selection

1: **Input:** round_cnt, label (shape: $[m, n]$), seed (shape: $[p]$), edge_num
2: **Output:** interested_indices, rand_seed1, weighted
3: **if** $\left(\frac{\text{round\_cnt}}{2} \mod 2\right) = 3$ **then**
4: $interested\_indices \leftarrow$ random choice from range($label.shape[1], edge\_num$, replace=True)
5: $rand\_seed1 \leftarrow$ random choice from range($seed.shape[0], edge\_num$, replace=True)
6: $weighted \leftarrow$ False
7: **end if**
8: **Return:** interested_indices, rand_seed1, weighted

This model uses supervised learning with backpropagation to minimise a multi-task loss function, with task weights adjusted based on significance. Training parameters, such as epochs, optimiser choice, learning rate, and hyperparameters, follow the baseline model, which is optimised for performance, except for a correction to the MTFuzz code.

We identified an issue in the Github repository for the MTFuzz project (specifically concerning the nn.py code file) [52] that we document here. Algorithm 1 illustrates whether to use rare edge selection in the original code of MTFuzz. There is a logical inconsistency within the "if" statement. Specifically, the expression `if (int(round_cnt/2) % 2) == 3`, where `round_cnt` serves as the iteration counter, is logically flawed. The modulus operation %2 only yields results of either 0 or 1, making it impossible for the expression to evaluate to 3. To correct this, we modified the condition in our experiments to: `if (int(round_cnt/2) % 2) == 1`.

For completeness, we performed a comparative analysis between the original and modified statements to evaluate the fuzzing performance, with `readelf` serving as the software target. Whilst both approaches achieved similar edge coverage in our testing, we chose to conduct our further experimentation with the corrected code `if (int(round_cnt/2) % 2) == 1` to ensure a robust approach to our testing methodology.

3.2. Our proposed solution

Fig. 1 illustrates the overall workflow of the proposed approach. While the initial phase, which involves training the multitask feed-forward neural network, closely follows the method used in MTFuzz, our approach introduces a new technique for training the network to generate sparse test cases. These test cases are designed to focus on parts of the input data that, when altered, have the potential to trigger different branches or execution paths in the software—i.e., the input bytes most likely to affect code coverage. This approach is referred to as *Leveraged MTFuzz* (LMTFuzz), emphasising the use of activation layers and preventing the issue of dying nodes through LeakyReLU, a method specifically intended to address this challenge.

In the following sections, we present the mathematical foundations of our approach and each solution, emphasising their direct impact as observed in both individual and combined testing.

3.3. LReLU vs. ReLU activation functions

The activation functions ReLU and LReLU are essential for incorporating non-linearity into neural networks, enabling them to capture intricate relationships within data. ReLU is frequently favoured due

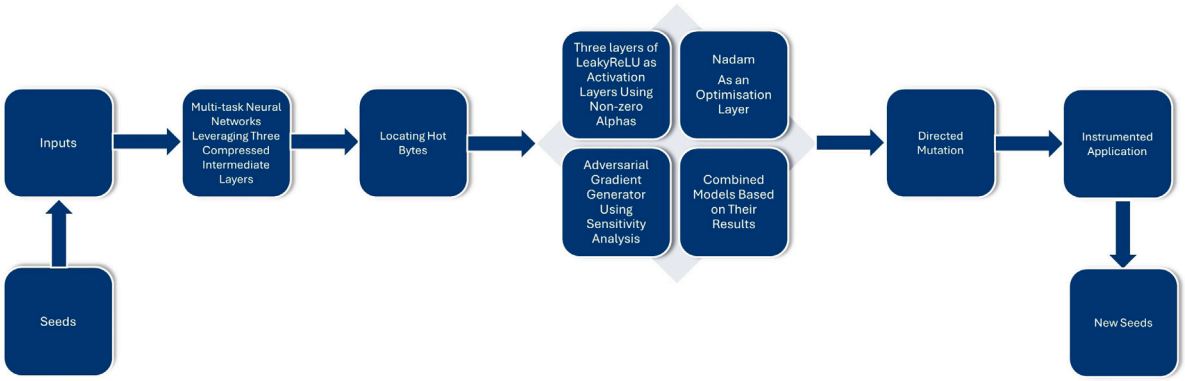

Fig. 1. Workflow of LMTFuzz.

to its ability to streamline gradient calculations, enhance training efficiency, and yield favourable results across various applications. The ReLU activation function [53,54] is defined as:

$$f(x) = \max(0, x) \tag{1}$$

In ReLU, the output is 0 for negative inputs and equal to the input for positive inputs. This can cause the "dying ReLU" problem, where neurons become inactive when the input is negative or updates lead the neuron to remain in the negative regime, resulting in:

$$\frac{\partial f(x)}{\partial x} = 0, \quad \text{for } x < 0 \tag{2}$$

This prevents neurons from updating during backpropagation; specifically, neurons with negative pre-activation ($x < 0$) do not contribute to gradient updates, resulting in a *sparse gradient flow*. This can hinder model training, particularly when features associated with negative activations are important. In contrast, LReLU [31] addresses this limitation by permitting a small, non-zero gradient for negative inputs. The LReLU activation function introduces a small slope for negative inputs to alleviate this issue:

$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha x, & \text{if } x \leq 0 \end{cases} \tag{3}$$

where α is a small positive constant. For LReLU, the gradient is:

$$\frac{\partial f(x)}{\partial x} = \begin{cases} 1, & \text{if } x > 0 \\ \alpha, & \text{if } x \leq 0 \end{cases} \tag{4}$$

Analysing the differences between LReLU and ReLU is key to identifying which enhances model performance in terms of convergence, stability, and accuracy. LReLU plays a critical role in both classification and regression tasks, as discussed in the following subsections.

3.3.1. Classification loss and LReLU

Classification tasks in MTFuzz, uses *binary cross-entropy loss*. For binary classification, the loss is defined as:

$$\mathcal{L}_{\text{BCE}} = -\sum_i y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i) \tag{5}$$

where y_i is the true label, and $\hat{y}_i$ is the predicted probability from the **sigmoid** output.

Gradient Analysis for Classification: For a classification network with LReLU activations, the gradient of the loss $\mathcal{L}_{\text{BCE}}$ with respect to an intermediate LReLU activation x is:

$$\frac{\partial \mathcal{L}_{\text{BCE}}}{\partial x} = \frac{\partial \mathcal{L}_{\text{BCE}}}{\partial f(x)} \cdot \frac{\partial f(x)}{\partial x} \tag{6}$$

Therefore, unlike ReLU, LReLU allows gradients to propagate through inactive neurons by maintaining a small but nonzero derivative ($\frac{\partial f(x)}{\partial x} = \alpha$ for $x < 0$). This prevents *vanishing updates*, ensuring continuous feature learning and mitigating the "dead neuron" problem. By preserving gradient flow, LReLU enhances convergence stability and enables the network to extract meaningful features even from negative inputs, improving *edge- and context-sensitive* representations.

3.3.2. Regression loss and LReLU

For regression tasks, a common loss function is *Mean Squared Error (MSE)*:

$$\mathcal{L}_{\text{MSE}} = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \tag{7}$$

where y_i is the true value and $\hat{y}_i$ is the predicted output. If a neuron enters the negative regime (i.e., $x < 0$), the gradient is zero ($\frac{\partial f(x)}{\partial x} = 0$), which means no weight updates will occur. This leads to slower learning, especially in tasks like *approach-sensitive coverage*, where negative activations can carry valuable information. In contrast, the small negative slope (α) in LReLU ensures that even for negative values of x, the neuron will still contribute to the gradient, improving the *smoothness* of the loss landscape and accelerating convergence.

Gradient Analysis for Regression: The gradient of the MSE loss with respect to an LReLU activation x is:

$$\frac{\partial \mathcal{L}_{\text{MSE}}}{\partial x} = 2(y - \hat{y}) \cdot \frac{\partial f(x)}{\partial x} \tag{8}$$

For negative x, the gradient is scaled by α, ensuring that the model receives *stable weight updates* even for inactive neurons. This leads to more reliable learning and better *approach-sensitive coverage*, where even neurons that are "inactive" during certain phases of training still contribute meaningfully to the model's updates. Therefore, from a loss function perspective, LReLU plays a significant role in both classification and regression tasks.

3.4. Nadam vs. Adam optimiser

The Adam optimiser is widely acknowledged for its adaptive learning rate, which combines the strengths of Root Mean Square Propagation (RMSProp) and SGD with momentum, thereby facilitating faster convergence. In contrast, the inclusion of Nesterov momentum introduces a subtle anticipation of gradient updates, which can further optimise performance. While Adam's adaptive learning rate and integration of RMSProp with SGD momentum allow for faster convergence compared to SGD, Nadam further enhances these capabilities by incorporating Nesterov momentum. Research by Dozat [32] demonstrates that Nadam outperforms Adam in terms of both accuracy and loss reduction during training, positioning it as a promising alternative in some applications beyond fuzzing. In the following subsections, we explore the mathematical foundation of Nadam to assess whether it may improve fuzzing performance.

1. Gradient Computation (this stage remains the same as in Adam): The initial phase involves calculating the gradient of the loss function with respect to the model parameters, a vital process that is also utilised in the Adam optimisation algorithm.

$$g_t = \nabla_\theta J_t(\theta_t) \tag{9}$$

where g_t is the gradient of the loss function J_t with respect to the parameters θ_t at time step t, with respect to the shared parameters θ. The combined gradient for the shared layers is then computed by aggregating the per-task gradients, optionally weighted by a task-specific importance factor w_t:

$$g_{\text{shared}} = \sum_{t=1}^{T} w_t \cdot g_t \tag{10}$$

where T is the total number of tasks and w_t represents a task-specific weighting factor.

2. Nesterov-accelerated First Moment based on Exponential Moving Averages of the Gradient: To enhance momentum and accelerate convergence, Nadam applies Nesterov acceleration to the first moment estimate. The key distinction between Adam and Nadam lies in how the first moment (momentum) is computed. In Adam, the momentum term is updated using a conventional exponential moving average of gradients, given by:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \tag{11}$$

where β_1 represents the decay rate for the first moment, commonly referred to as momentum, and is typically set close to 1 (e.g., 0.9). In the case of shared layers, the first moment can be aggregated using a weighted sum:

$$m_{\text{shared}} = \sum_{t=1}^{T} w_t \cdot m_t \tag{12}$$

However, Nadam introduces Nesterov acceleration by incorporating a lookahead mechanism, modifying the first moment estimate as follows:

$$\hat{m}_t = \frac{\beta_1 m_t + (1-\beta_1) g_t}{1-\beta_1^t} \tag{13}$$

For shared layers, the corrected first moment is aggregated as:

$$\hat{m}_{\text{shared}} = \sum_{t=1}^{T} w_t \cdot \hat{m}_t \tag{14}$$

This formulation anticipates future updates by integrating the gradient into the momentum component, improving convergence.

3. Second Moment (this stage remains the same as in Adam): The second moment, which quantifies the variance of gradients, is computed similarly to Adam's approach:

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \tag{15}$$

where β_2 is the decay rate associated with the second moment, a component inherited from the RMSProp algorithm, typically set to around 0.999. For shared layers, the second moment can be aggregated across tasks as:

$$v_{\text{shared}} = \sum_{t=1}^{T} w_t \cdot v_t \tag{16}$$

4. Bias Correction: In the early stages of Adam optimisation, the moving averages of the gradients tend to be biased towards zero due to their initialisation at zero. This bias can distort the estimates of the first and second moments, particularly during the initial time steps. To mitigate this bias, Adam employs bias-correction formulas to provide more accurate estimates. The bias-corrected estimates for the first and second moments are given by:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \tag{17}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t} \tag{18}$$

These formulas correct the estimates of the first moment (m_t) and second moment (v_t) by adjusting for the bias introduced by their initial values, particularly during the early time steps. In the case of Nadam, the bias correction is similarly applied to the second moment for shared layers as follows:

$$\hat{v}_{\text{shared}} = \sum_{t=1}^{T} w_t \cdot \hat{v}_t \tag{19}$$

This formulation helps improve the accuracy of the parameter updates by compensating for the early bias in the second moment estimate. The bias correction ensures that the optimisation process remains stable and efficient throughout training.

5. Parameter Update: The parameter update equation in Nadam follows a similar structure to Adam but with adjustments to the first moment. The update equation is:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t}+\epsilon}\hat{m}_t = \theta_t - \frac{\eta}{\sqrt{\hat{v}_{\text{shared}}}+\epsilon}\hat{m}_{\text{shared}} \tag{20}$$

where η is the learning rate and ϵ is a small constant for numerical stability. Nadam improves convergence by incorporating a look-ahead gradient calculation. The key difference from Adam lies in the update method for the momentum term. By integrating gradients and momentum across tasks, Nadam optimises shared layers in multi-task learning, offering faster convergence and computational efficiency for complex targets.

3.5. Optimal layer selection through sensitivity analysis in the multiple tasks environment

Sensitivity analysis is a valuable technique for identifying the layers in a neural network that most influence output variations. This method plays a crucial role in optimising the selection of layers for gradient calculations, which are important for tasks like backpropagation and saliency mapping. By focusing on key layers, it reduces computational overhead and improves model interpretability. The goal is to fine-tune the most influential parameters, enhancing overall model performance and ensuring more targeted gradient updates. This process helps adjust gradient propagation based on layer sensitivity, leading to better training dynamics and enhanced model robustness. The following subsections outline the key steps of the methodology: a formal definition of layer sensitivity based on gradient magnitudes, a mathematical relationship between sensitivity scores and task performance, gradient aggregation across multiple tasks, a theoretical formulation of the impact of sensitivity analysis on gradient flow during training, and fuzzing behaviour.

3.5.1. Formal definition of layer sensitivity

Layer sensitivity can be formally defined as the gradient of the model's loss with respect to the output of a particular layer, which measures the responsiveness of that layer to small perturbations in the input. Mathematically, the sensitivity S^l of a layer l can be expressed as:

$$S^l = \text{mean}\left(\left|\frac{\partial \mathcal{L}}{\partial \mathbf{O}^l}\right|\right) \tag{21}$$

where $\mathcal{L}$ is the loss function, and $\mathbf{O}^l$ represents the output of layer l. This sensitivity score quantifies how much the output of a layer changes with respect to changes in the model's input, providing a measure of the layer's importance in the model's decision-making process.

3.5.2. Sensitivity scores and task-specific performance

To align sensitivity analysis with task-specific performance, we propose a relationship between the sensitivity scores and the model's performance on a given task. This relationship helps in adjusting the learning process such that more sensitive layers have a greater impact on optimising the model for the task at hand. The task-specific performance $\mathcal{P}$ can be related to the sensitivity scores by the weighted sum:

$$\mathcal{P} = \sum_{l=1}^{n} w^l \cdot S^l \tag{22}$$

where w^l is a task-specific weight that reflects the importance of layer l for a particular task, and S^l is the sensitivity score of that layer. This relationship ensures that layers with higher sensitivity scores contribute more to the overall performance, directly influencing the model's optimisation process in a way that reflects task-specific needs.

3.5.3. Gradient aggregation over multiple tasks

In multi-task learning scenarios, it is essential to aggregate gradients across tasks while maintaining the influence of sensitivity scores. To achieve this, we aggregate gradients across layers by computing the gradient of the loss with respect to the model's parameters, taking into account both task-specific weights and layer sensitivities. The aggregated gradient ∇W^l_{agg} for each layer l is given by:

$$\nabla W^l_{\text{agg}} = \sum_{t=1}^{T} w^l_t \cdot \nabla W^l_t \tag{23}$$

where T is the number of tasks, w^l_t is the task-specific weight for layer l under task t, and ∇W^l_t is the gradient of the loss with respect to the weights of layer l for task t. This formulation ensures that gradients from all tasks are weighted according to their relevance and the sensitivity of each layer, allowing for efficient multi-task learning.

3.5.4. Impact on gradient flow during training

The incorporation of sensitivity analysis into the gradient flow modifies the backpropagation process by prioritising layers with higher sensitivity scores. This prioritisation affects the weight updates, where the update for the weights W^l of layer l is adjusted by a factor proportional to its sensitivity:

$$\Delta W^l = -\eta \cdot S^l \cdot \nabla_{W^l} \mathcal{L} \tag{24}$$

Here, η is the learning rate, and $\nabla_{W^l} \mathcal{L}$ is the gradient of the loss function with respect to the weights of layer l. The term S^l acts as a scaling factor that controls the magnitude of the update for layers with higher sensitivity, ensuring that these layers have a greater influence on the optimisation process. This modification to the gradient flow encourages faster convergence in sensitive layers and enhances the overall stability of the training process.

3.5.5. Sensitivity scores and fuzzing behaviour

Fuzzing behaviour refers to the process of generating adversarial perturbations to test the robustness of the model. Sensitivity analysis plays a crucial role in this process by identifying layers that are more sensitive to small changes in input, which are most susceptible to adversarial manipulation. To generate adversarial examples, we compute the gradients of the loss with respect to the model's input data, taking into account the sensitivity of each layer. The total gradient $\nabla \mathcal{L}$ is computed as:

$$\text{total\_grads} = \sum_{l=1}^{n} S^l \cdot \nabla W^l \tag{25}$$

This aggregated gradient reflects the combined influence of each layer's sensitivity on the overall model. The adversarial perturbations are then generated by manipulating the input data in the direction of the most influential gradients, which are determined by the sensitivity scores. Layers with higher sensitivity scores are more likely to contribute significantly to the adversarial perturbations, resulting in more targeted and effective attacks. This relationship between sensitivity and fuzzing behaviour allows for more efficient adversarial training and model robustness evaluation.

3.5.6. Task-specific weighting of gradients based on sensitivity

Finally, in a framework that incorporates sensitivity analysis, task-specific weighting of gradients is necessary to prioritise layers that are more relevant to specific tasks. The weight for each layer l under task t is determined by the sensitivity score:

$$w^l_t = \frac{S^l_t}{\sum_{l'} S^{l'}_t} \tag{26}$$

This ensures that layers with higher sensitivity scores, which are more influential for the task at hand, will contribute more to the gradient updates during training. By assigning appropriate task-specific weights, the model can be optimised more effectively for each task, while still ensuring that sensitive layers are prioritised across the entire training process.

Algorithm 2 Sensitivity Analysis Algorithm

```
Input: model, input_data
Output: sensitivity_scores
sensitivity_scores = {}
for each layer in model.layers do
    if layer has output then
        layer_name = layer.name
        layer_output_func = create function to compute the output of layer
        layer_output = layer_output_func(input_data)
        sensitivity_score = mean of absolute values of layer_output
        sensitivity_scores[layer_name] = sensitivity_score
    end if
end for
Return: sensitivity_scores
```

3.5.7. Algorithmic representation of sensitivity analysis and adversarial attack generation

To formalise this process algorithmically, the `sensitivity_analysis` presented in Algorithm 2 computes sensitivity scores for each layer in a neural network model, offering a measure of how each layer's output responds to variations in the input data through the following phases:

1. **Start:** Begin the process of layer analysis.
2. **Aggregate Gradients from Multiple Layers:** Collect gradient information from different layers to get a comprehensive view of their contributions.
3. **Analyse Intermediate Layers:** Examine how intermediate layers contribute to the network's predictions.
4. **Perform Sensitivity Analysis:** Systematically assess how changes in each layer affect the output, identifying key layers.
5. **Identify "high-sensitivity layers":** Determine which layers are most influential for fine-tuning and model optimisation.

This process helps identify which layers are most influential in the model's behaviour. The function takes two inputs: the neural network model (`model`) and the data used for analysis (`input_data`). For each layer in the model, the function checks if the layer has an output. If so, it creates a Keras function that computes the layer's output for the given input data. After obtaining the output, the function calculates the sensitivity score for that layer by computing the mean of the absolute values of the layer's output. The sensitivity score reflects the layer's responsiveness to the input data, where higher values indicate that small changes in the input lead to more significant changes in the output of that layer.

The function then returns a dictionary where the keys are the layer names and the values are their corresponding sensitivity scores. This analysis allows for a deeper understanding of the contribution of each layer to the overall model behaviour, particularly in adversarial attack generation. By identifying layers that are more sensitive to input changes, this method can help focus attacks on those layers, leading to more targeted and effective adversarial attacks. This strategy provides a more detailed and rigorous evaluation than deterministic models that select layers without such analysis.

Algorithm 3 Adversarial Attack Samples Generation Using Sensitivity Analysis

```
Input: list_of_seed_indices, model, layer_list, seed
Output: adversarial_list
adversarial_list = {} #Initialise adversarial examples list
total_grads = None # Initialise gradient accumulation
input_data = reshape seed[list_of_seed_indices[0]] to shape (1, seed.shape[1]) #Reshape seed for model input
sensitivity_scores = sensitivity_analysis(model, input_data) # Compute sensitivity scores for layers by Algorithm 2
sorted_layers = sort layer_list by sensitivity_score in descending order # Sort layers by sensitivity
for (layer_name, layer) in sorted_layers do
    if layer has output then
        loss = layer.output[:, random_index(layer.output.shape[-1])] # Choose random output for loss
        grads = gradients of loss w.r.t model.input # Compute gradients
        iterate = create function for loss and grads # Create gradient evaluation function

        for index in range(len(list_of_seed_indices)) do
            x = reshape seed[list_of_seed_indices[index]] to shape (1, seed.shape[1]) # Reshape each seed
            loss_value, grads_value = iterate([x]) # Evaluate gradients
            if total_grads is None then
                total_grads = abs(grads_value) # Initialise gradient accumulation
            else
                total_grads += abs(grads_value) # Accumulate gradients
            end if
        end for
    end if
end for
influential_indices = flip argsort(total_grads, axis=1)[-MAX_FILE_SIZE:] # Get most influential feature indices
val = sign of total_grads[0][influential_indices] # Determine perturbation direction
for index in range(len(list_of_seed_indices)) do
    adversarial_list.append((influential_indices, val, seed[list_of_seed_indices[index]])) # Store adversarial example
end for
Return: adversarial_list # Return the list of adversarial examples
```

Algorithm 3 presents a framework for generating adversarial attack samples within the baseline MTFuzz [30]. This framework incorporates a modified sensitivity analysis, which we developed and integrated to enhance the attack generation process by leveraging sensitivity scores. This algorithm details the computation of sensitivity scores for each layer in the model and demonstrates how these scores are leveraged to generate adversarial perturbations.

This sensitivity-aware approach provides a more nuanced strategy compared to deterministic models that select layers arbitrarily without such evaluation. It allows for a more targeted and effective manipulation of gradients in adversarial training, ultimately contributing to the development of more robust models. By focusing on the layers that contribute most significantly to model performance (as identified through sensitivity analysis), adversarial attacks can be optimised to maximise their impact.

3.6. Combined testing scheme based on selection of best model

Integrating LReLU with sensitivity analysis or LReLU with Nadam is expected to enhance the training process, boost model generalisation, and increase execution path diversity, ultimately leading to a more efficient fuzzing process and improved edge coverage. This is particularly beneficial in scenarios where dead neurons caused by gradient vanishing are problematic or when faster convergence and optimal layer selection are critical. LReLU effectively mitigates gradient vanishing, Nadam accelerates convergence with momentum-based updates, and sensitivity analysis optimises parameter weighting. The overall payoff $\mathcal{U}$ in terms of edge coverage is modelled as a function of the contributions from LReLU (λ), Nadam (ν), and Sensitivity Analysis (σ):

$$\mathcal{U} = f(C; \lambda, \nu, \sigma) \tag{27}$$

where f represents the combined effect of these techniques on maximising edge coverage.

This formulation ensures that optimising aligns with selecting the best combination of activation functions (LReLU vs. ReLU), optimisers (Nadam vs. Adam), and sensitivity analysis techniques, ultimately leading to an improved fuzzing framework.

4. Experimental framework

This section outlines the fuzzing experiment configuration, including the benchmark, testing environment, and hyperparameters.

4.1. Software targets under test

Fuzzing targets refer to the specific system components or software being tested through fuzzing. These targets can span various domains, such as file formats, network protocols, APIs, embedded systems, or web applications. For this study, we conducted our experiments on six linux software tools: `djpeg`, `mutool`, `size`, `nm`, `hb-fuzzer`, and `readelf`. These tools come from different software packages, with only `size`, `nm`, and `readelf` being extracted from `binutils-2.30`. The categorisation of these tools is shown in Table 1. The selected target tools (djpeg, mutool, size, nm, hb-fuzzer, readelf) are builtin targets within MTFuzz. Since the original MTFuzz paper includes these targets, we chose to maintain them to ensure direct comparability with the baseline results presented in the original work. This allows readers to easily assess how our approach performs relative to MTFuzz and other state-of-the-art fuzzers that the baseline has already been shown to outperform.

4.2. Testing environment

To assess the impact of resource allocation on MTFuzz's performance, we conducted fuzzing tests over a 24-h period. The experiments were conducted on a desktop workstation running VMware Workstation 17 on Windows 11 Pro, utilising six virtual machines (VMs) for six targets running Kali Linux, each with 8 GB of RAM allocated. Each Kali VM had access to 80 GB of disk space and shared CPU resources from the Windows host, ensuring sufficient resources for the fuzzing tasks. This configuration was chosen to simulate a distributed environment and investigate how multi-task learning scales with parallel task execution during fuzzing. Each of the baseline MTFuzz and three LMTFuzz versions was run on identical VM snapshots, with the six parallel VMs representing testing on six different targets. All conditions were kept consistent across all fuzzers to ensure reliable and comparable results.

The host machine was equipped with 10 CPU cores and 64 GB of RAM to meet the parallelism and memory demands of the virtualised setup, with at least one core logically dedicated to each VM. Additionally, the host machine had access to at least 3 terabytes of storage space, ensuring ample capacity for storing the data generated during the experiments. This setup was designed to prevent bottlenecks and ensure a precise evaluation of the impact of resource allocation on gradient calculation within multi-task learning.

4.3. Activation and optimisation hyperparameters

We define the specific parameters and configurations used in our fuzzing experiments to ensure reproducibility and accuracy. First, we detail the activation function parameters, with particular emphasis on LReLU. For this activation function, we set the slope of the negative region to $\alpha = 0.01$. This choice of $\alpha = 0.01$ was determined through preliminary experiments, which demonstrated that it provided optimal performance in terms of edge coverage and overall fuzzing effectiveness, outperforming other α values such as 0.1, 0.2, 0.3, and 0.4. This value follows conventional practices, where minor negative values are allowed to pass through the activation function, thus mitigating issues associated with vanishing gradients. While further optimisation of α is a promising direction for future research, this study primarily focuses

Table 1
Command-line tools used for processing different file types, along with their corresponding source packages and descriptions.

Input	Tool	Source Package	Description
JPEG image	djpeg	libjpeg-9c	A command-line tool to decompress JPEG images.
PDF file	mutool	MuPDF-1.12.0	A tool from MuPDF for working with PDF documents.
Object file	size	binutils-2.30	Displays the size of sections in object files.
	nm -C	binutils-2.30	Lists symbols from object files with demangling of C++ symbols.
Text file	hb-fuzzer	Harfbuzz-1.7.6	A tool for fuzz-testing the Harfbuzz text shaping engine.
ELF file	readelf -a	binutils-2.30	Displays detailed information about ELF (Executable and Linkable Format) files.

on evaluating the impact of LReLU's non-zero gradient on fuzzing performance.

Further, we outline the optimisation algorithms employed in the experiments, specifically Adam and Nadam. Both optimisers are initialised with a learning rate coefficient, $\eta = 0.001$. This learning rate was intentionally retained to match the value used in MTFuzz's implementation of Adam, ensuring a fair comparison between the two optimisation strategies. By keeping η constant, we are able to isolate and evaluate the impact of the optimisation strategy itself without confounding it with variations in the learning rate. Additionally, the choice of $\eta = 0.001$ is commonly used in the literature, as it strikes a balance between efficient convergence and stability across different experimental iterations.

5. Evaluation

In this section, we begin by observing the experimental results obtained from the methods and optimisers used in the baseline model of MTFuzz, as presented in Table 2. These results are then compared with those of the baseline model to assess performance differences. Subsequently, we examine the impact of activation functions, optimisation techniques, and their influence on training, accuracy, and fuzzing performance. Furthermore, Table 3 provides detailed explanations of each column in Table 2, offering insights into their role and significance in evaluating the performance of ML models.

Our study isolates and evaluates the individual effects of three key components—LReLU, Nadam, and sensitivity analysis. By varying one component at a time while keeping the others constant, we assess the unique contribution of each to overall performance. Subsequently, we combine Nadam and sensitivity analysis with LReLU, focusing on their integration due to LReLU's effectiveness in preventing dead neurons. This approach aims to improve metrics such as edge coverage, maximum accuracy, and minimum loss. By evaluating each component both independently and in combination with LReLU, we ensure a thorough analysis of their collective impact.

To understand variations in model performance, we analyse six key indicators: Maximum Accuracy, Loss at Maximum Accuracy, Iteration for Maximum Accuracy, Minimum Loss, Accuracy at Minimum Loss, and Iteration for Minimum Loss. These metrics help identify the training iteration where the model achieves its lowest loss, which may differ from the iteration of peak accuracy.

Each column in Table 2 plays a critical role in assessing the performance of various ML models, offering a comprehensive view of models strengths and areas for improvement compared to baseline model of MTFuzz. We observe both maximum accuracy and minimum loss metrics separately. This is due to the fact that Accuracy measures the proportion of correct predictions, while loss quantifies how well the model fits the data. As a result, the point at which accuracy reaches its maximum may not necessarily coincide with the point where the loss is minimised, leading to different values for each metric and the iteration in which they occur.

Therefore, while maximum accuracy and minimum loss may align at times, they often occur at different iterations due to their distinct progress during training. Achieving lower loss is crucial for improving model performance, as it typically indicates better generalisation and more efficient learning, particularly in fuzzing, where it enhances the model's ability to detect vulnerabilities. This structure allows for easy comparison, helping to identify which model performs best under various conditions.

In the following subsections, we evaluate the methods used in the baseline model through comparative analyses of different activation functions, optimisation techniques, and a sensitivity analysis, exploring their impact on training, accuracy, and fuzzing performance.

5.1. Max accuracy

Maximum accuracy in DNN-enabled fuzzing refers to the ability of deep neural networks (DNNs) to precisely identify vulnerabilities and generate effective test inputs. It represents the highest level of accuracy a DNN can achieve in predicting fuzzing-related outcomes. In this context, LReLU and its variant, Sensitivity with LReLU, consistently outperform baseline models across most evaluation tools (four out of six). Furthermore, LReLU, its sensitivity-enhanced variant, and Sensitivity itself collectively demonstrate improved performance in five out of six assessments on `djpeg`, `mutool`, `size`, `nm`, and `readelf`. This finding underscores the effectiveness of integrating LReLU and sensitivity analysis in significantly improving accuracy across various analytical tools.

The performance of LReLU and its sensitivity-enhanced variant did not improve in the case of `hb-fuzzer`. In this specific target, performance marginally worsened. This variability can be attributed to several factors. First, different fuzzing targets may not always align with the strengths of a particular model. For example, complex targets like `hb-fuzzer` may exhibit lower sensitivity to activation function changes due to inherent noise or complexity. Additionally, the effectiveness of activation functions such as LReLU depends on the fuzzing environment's ability to support gradient propagation. In environments with lower gradient sensitivity, such as `hb-fuzzer`, further optimisation may have less impact. Other influencing factors include model architecture, training data, and hyperparameters such as learning rates and optimisation algorithms. Fine-tuning these elements could enhance performance across more targets, emphasising the importance of adapting DNN-based fuzzing strategies to specific target characteristics.

5.2. Loss metrics

To provide a more comprehensive evaluation, we present both maximum accuracy and minimum loss values, along with their corresponding pairs. For further details, please refer to the columns labeled `Loss at Maximum Accuracy` and `Minimum Loss` or the `Max Accuracy` and `Accuracy at Min Loss` for comparison.

As shown in Table 2, the models focusing on minimum loss, including LReLU, Sensitivity, and their combined variant (Sensitivity with

Table 2
Comparison of models across different tools.

Model	Total Iterations	Max Accuracy	Iteration of Max Accuracy	Loss at Max Accuracy	Min Loss	Iteration of Min Loss	Accuracy at Min Loss	Max Edge Coverage	Average Total Executions	Max Dimension (pair)	Unique Bugs
djpeg											
Baseline	10	0.8849	10	0.1027	0.1027	10	0.8849	2375	3873131.4	(2418, 1374)	376
LReLU	12	**0.8887**	12	0.0953	**0.0953**	12	0.8887	2385	3845286.3	(2664, 1431)	**591**
Nadam	11	0.8217	11	0.1359	0.1353	9	0.816	**2391**	3856704.9	(2695, 1463)	423
Hybrid	12	0.8559	11	0.1184	0.1184	11	0.8559	**2391**	4100812.7	(2644, 1438)	265
Sensitivity	9	0.8835	9	0.0985	0.0985	9	0.8835	2389	3916204.8	(2508, 1393)	178
Sensitivity with LReLU	9	0.8886	9	0.0978	0.0978	9	0.8886	2365	3951695.9	(2464, 1392)	308
mutool											
Baseline	6	0.9606	5	0.0287	0.0287	5	0.9606	5357	3545334.7	(2603, 1962)	73
LReLU	9	0.9613	2	0.0309	0.0275	6	0.9613	**5563**	4037122.0	(2699, 2041)	116
Nadam	9	0.9331	7	0.0493	0.0481	6	0.9324	5468	3619220.7	(2783, 2014)	140
Hybrid	9	0.9439	4	0.0419	0.0419	4	0.9439	5431	3640603.9	(2707, 2026)	126
Sensitivity	8	**0.9659**	6	0.0249	**0.0244**	8	0.9659	5412	3390770.4	(2718, 2034)	122
Sensitivity with LReLU	8	0.9627	4	0.0280	0.0267	8	0.9626	5527	3470468.3	(2742, 2040)	**154**
size											
Baseline	9	0.9108	1	0.1499	0.0709	8	0.8578	5175	5705240.7	(2318, 2402)	**4**
LReLU	13	0.9146	1	0.1453	**0.0590**	13	0.8427	6135	6248035.7	(2299, 2744)	**4**
Nadam	14	0.807	9	0.0896	0.0703	14	0.806	**6222**	6102337.7	(2636, 3010)	2
Hybrid	13	0.8333	7	0.0937	0.0683	13	0.8187	6161	6161699.8	(2349, 2814)	2
Sensitivity	8	0.9090	1	0.1487	0.0668	8	0.8633	4923	5358260.4	(2056, 2304)	0
Sensitivity with LReLU	8	**0.9147**	2	0.1152	0.0693	8	0.8661	4972	4709890.4	(1959, 2217)	3
nm											
Baseline	7	0.9129	3	0.1020	0.0741	7	0.8647	6762	6208137.2	(2679, 3175)	6
LReLU	9	0.9038	3	0.1166	0.0735	9	0.8472	7886	5867734.0	(3185, 4063)	1
Nadam	9	0.8338	5	0.1119	0.1119	5	0.8338	7918	5827720.0	(3183, 4023)	2
Hybrid	9	0.8770	4	0.0900	0.0763	8	0.8401	8183	5652667.2	(3421, 4358)	**22**
Sensitivity	6	0.9299	3	0.0794	**0.0477**	6	0.9031	**8378**	4623163.3	(3725, 4527)	12
Sensitivity with LReLU	6	**0.9289**	3	0.0794	0.0482	6	0.9016	8234	4603283.5	(3723, 4404)	0
hb-fuzzer											
Baseline	6	**0.9191**	2	0.079	**0.0622**	4	0.8979	8899	6036131.6	(5744, 4462)	0
LReLU	10	0.9178	2	0.0799	0.0645	4	0.8917	**9493**	6137737.6	(6776, 5032)	0
Nadam	11	0.8923	2	0.096	0.074	4	0.8734	9071	5521524.5	(5866, 4586)	0
Hybrid	10	0.8868	2	0.0996	0.0757	4	0.8628	9152	5549557.3	(6203, 4696)	0
Sensitivity	6	0.9085	2	0.0839	0.0663	4	0.893	8798	5604992.0	(5398, 4344)	0
Sensitivity with LReLU	6	0.9102	2	0.0827	0.0669	4	0.8843	8963	5534210.8	(5381, 4484)	0
readelf											
Baseline	6	0.8931	1	0.1534	0.0928	2	0.8719	10 134	5192164.8	(12838, 6480)	**6313**
LReLU	10	**0.9178**	2	0.0799	0.0645	4	0.8917	**10 607**	6237030.0	(14094, 6965)	5980
Nadam	9	0.8728	1	0.1721	0.109	2	0.843	10 355	5121424.4	(13610, 6695)	894
Hybrid	9	0.8748	1	0.1707	0.1054	2	0.8536	10 070	5362083.7	(13737, 6546)	885
Sensitivity	6	0.878	2	0.0928	0.068	3	0.8444	9334	3418821.8	(13269, 6346)	1094
Sensitivity with LReLU	6	0.8998	1	0.1488	**0.0625**	6	0.7817	10 070	5643668.0	(13091, 6586)	5334

LReLU), consistently achieved the lowest loss rates across most targets. However, in the case of `hb-fuzzer`, the minimum loss rate and loss at maximum accuracy were comparable to, but not superior to, the baseline model.

In fact, the baseline model exhibited higher loss values for 5 out of 6 targets, underscoring a notable performance gap when compared to LReLU, Sensitivity, and Sensitivity with LReLU. Additionally, LReLU outperformed both Sensitivity and Sensitivity with LReLU on 3 out of 6 targets in terms of achieving the lowest loss rate. On the other hand, Sensitivity and the combined variant surpassed LReLU on 2 and 1 targets, respectively. Thus, we conclude that LReLU remains the best-performing model in terms of minimising loss.

5.3. Edge coverage

The results demonstrate that LReLU achieves consistently high maximum edge coverage across all models, indicating its effectiveness as a testing strategy. This robust performance suggests that LReLU is well-suited for applications requiring comprehensive edge coverage. One of the key reasons for its superior performance is the way LReLU allows for more effective exploration of the input space compared to other activation functions, making it particularly effective in identifying diverse edge cases during fuzz testing. While the Sensitivity model exhibited the lowest edge coverage specifically on the size target, it still performed comparably well on other targets, indicating its overall reliability.

LReLU, along with its combined variants, consistently secured top positions in edge coverage across a variety of targets. The superior edge coverage achieved by LReLU can be attributed to its smoother gradient properties, which help in better navigating the loss landscape and achieving more thorough exploration of the edge cases. Notably, LReLU achieved the highest edge coverage for three targets: `mutool`, `hb-fuzzer`, and `readelf`, underscoring its versatility and effectiveness across diverse applications. Furthermore, Nadam and its hybrid model incorporating LReLU both ranked equally in first place for the `djpeg` target, and first and second for the `size` target, highlighting the benefits of model integration for enhanced coverage, with LReLU consistently being a critical component of the top-performing configurations.

Moreover, LReLU maintained an impressive third position on the size target, following its hybrid version with Nadam, showcasing its competitiveness even when not in the top spot. Nadam's performance was also noteworthy, achieving the highest edge coverage for two targets – `djpeg` and `size` – suggesting that Nadam can serve as a robust alternative in specific contexts. This highlights that while LReLU is consistently a top performer, in certain scenarios, integrating other models, such as Nadam, can provide additional benefits.

Lastly, the Sensitivity model, along with its variant combined with LReLU, attained the highest edge coverage on the `nm` target. This finding emphasises the importance of exploring different model combinations to optimise edge coverage and suggests that further investigations into hybrid models may yield additional insights into enhancing testing effectiveness. The combination of Sensitivity with LReLU, in particular, seems to leverage the strengths of both models, providing enhanced edge coverage that would not be achievable by either model alone.

Table 3
Explanation of key metrics.

Column name	Description	Significance
Model	Lists the various ML models evaluated, each with different training techniques.	Comparing models helps assess performance differences and determine which configuration yields the best results.
Total Iterations	The number of training or testing iterations reached by each model within 24 h. The final iteration count may include both completed and incomplete iterations terminated at the 24-h mark.	More iterations may lead to a more thorough evaluation but risk overfitting.
Max Accuracy	The highest accuracy achieved by each model during evaluation, which does not necessarily occur at the iteration corresponding to the minimum loss.	Indicates the model's best performance, reflecting its ability to predict or classify correctly.
Iteration of Max Accuracy	The iteration during which the model achieved its maximum accuracy.	Helps in tracking model performance over time and assessing if further fine-tuning is necessary.
Loss for Max Accuracy	The loss rate associated with the model's highest accuracy.	Lower loss values at high accuracy suggest effective training and model robustness.
Min Loss	The lowest loss rate recorded during evaluation, providing insight into the model's error minimisation, which does not necessarily occur at the iteration corresponding to the maximum accuracy.	Provides insight into the model's generalisation capabilities and error minimisation.
Iteration of Min Loss	The iteration during which the model achieved its lowest loss.	Shows how the model's training progressed and whether further improvements are needed.
Accuracy for Min Loss	The accuracy corresponding to the minimum loss value recorded.	Assesses the model's balance between accuracy and error reduction.
Max Edge Coverage	Refers to the maximum code edge coverage during testing.	High edge coverage indicates thorough testing, contributing to improved fuzzing performance and potentially enabling the detection of more sophisticated software bugs.
Average Total Executions	The average number of test executions performed for each model.	Increased executions may enhance coverage and bug detection but also raise computational costs. Therefore, achieving better coverage with the lowest average number of total executions is crucial.
Max Dimension (pair)	The maximum dimensions of input data used during training or testing.	Helps to understand the data complexity and its influence on model learning, as well as the model's ability to capture sophisticated bugs. If it does not lead to issues such as overfitting, this complexity can be valuable for detecting more sophisticated bugs.
Unique Bugs	The number of unique bugs including unique crashes and unique hangs detected by the model during testing.	Indicates the effectiveness of the model in identifying issues in real-world software.

In conclusion, our analysis underscores the effectiveness of LReLU and its variants in achieving high edge coverage across multiple targets. It also highlights the potential of hybrid models for improving performance in specific scenarios. The comparative advantage of LReLU in these tests can be attributed to its ability to better explore edge cases, and the hybrid models further illustrate the potential for combining activation functions to achieve optimal coverage.

5.4. Discovery of new edges

In fuzzing, particularly when leveraging deep neural networks (DNNs), a "new edge" refers to the identification of a previously unexplored path or behaviour in the target program's control flow during testing. At each iteration of the experiment, new edge discovery signifies that the generated input has triggered a transition between basic blocks in the program's control flow graph (CFG) that had not been executed in prior runs. Since the total number of newly discovered edges is unique to each fuzzer for a specific iteration, tracking this metric over time reveals patterns of edge discovery, providing insight into the fuzzer's capability to explore deeper and more complex program states.

New edge discovery is a widely recognised indicator of a fuzzer's effectiveness and exploratory power over time, as a higher rate of unique edge discoveries suggests that the fuzzer is efficiently generating diverse and meaningful test cases. Over multiple iterations, analysing the rate at which new edges are discovered can highlight trends in exploration efficiency. For example, a fuzzer that maintains a consistently high rate of new edge discovery in the early stages demonstrates strong initial exploration capabilities, while one that continues discovering new edges in later stages exhibits long-term effectiveness in overcoming input saturation. Thus, evaluating new edge discovery patterns across iterations provides a quantifiable measure of both short-term and long-term fuzzing efficiency.

Fig. 2 presents the newly discovered edges identified through different methodologies over a 24-h period. The figures illustrate varying behaviours, generally showing a peak at the beginning, with some models requiring more iterations to reach the end of this duration. This initial peak can be attributed to the fact that, early in the fuzzing process, the model is likely to uncover a broader range of unexplored edges quickly. Sensitivity analysis demonstrates either an increase or a gradual decrease in newly discovered edges during the final iterations for five targets (e.g., `djpeg`, `size`, `nm`, `mutool`, and `readelf`). This indicates consistent edge discovery, with particularly strong performance observed for `nm` and an increase in new edge discovery during the final iterations for `readelf`, `size`, and `nm`, which is advantageous. The observed increase in new edge discovery in the

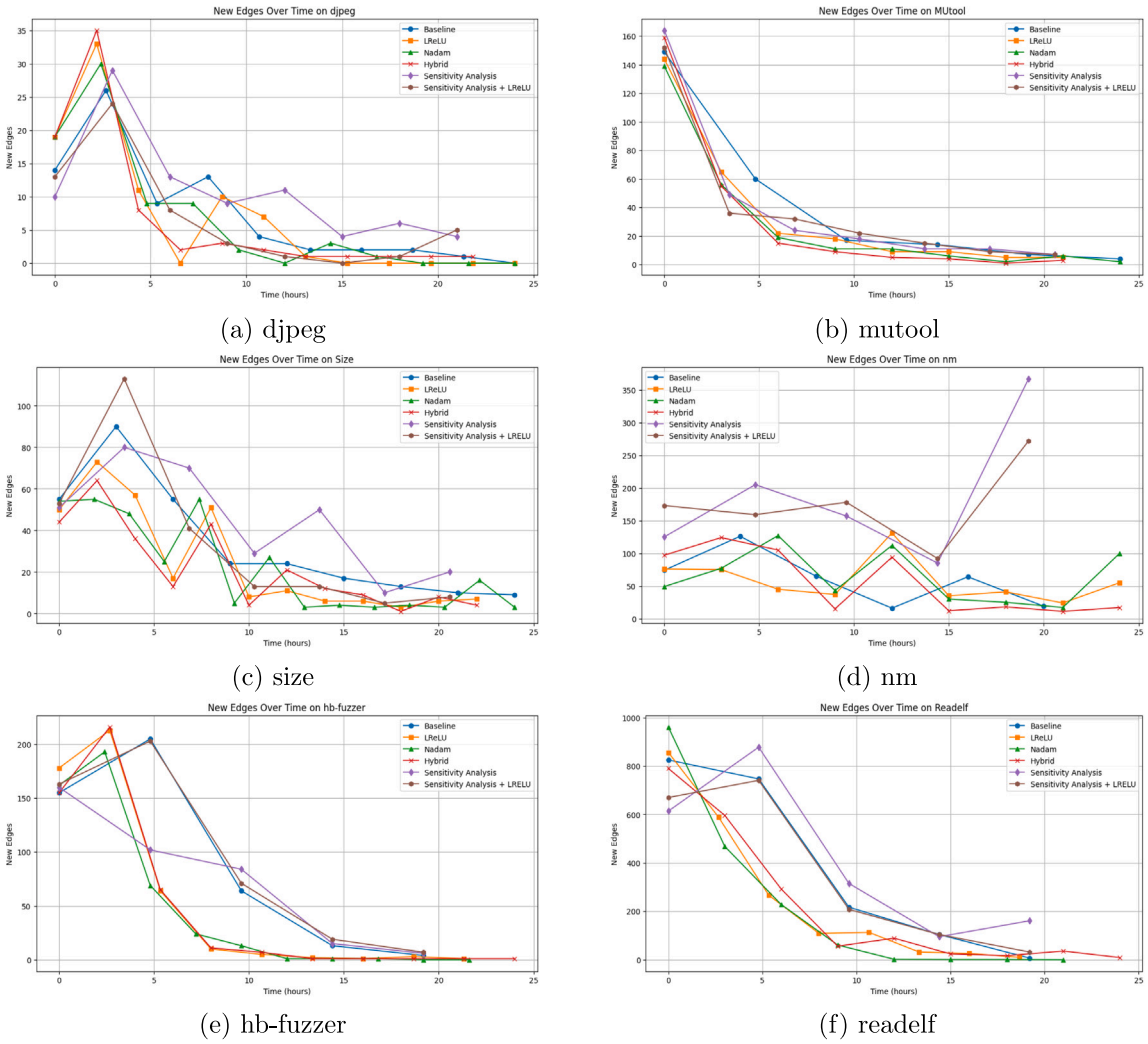

Fig. 2. New edge discovery patterns of different methodologies on various targets across iterations over time.
Note: The time slots are approximate, and some models did not discover new edges during the final hours.

final iterations for these targets suggests that the fuzzing model is beginning to uncover deeper, more complex paths in the control flow. These paths could involve more sophisticated code branches or corner cases that were less likely to be triggered during earlier iterations. For example, some complex conditions or specific input constraints could have only been met after repeated exploration, leading to the discovery of previously inaccessible control flow paths.

In other words, while most other methodologies exhibited a decreasing trend over this timeframe, sensitivity analysis on four targets – and its variant (sensitivity with LReLU) on a few targets (i.e., `nm`, `size`, and `djpeg`) – demonstrated a distinct pattern of new edge discovery towards the end of the period. This behaviour suggests that the sensitivity analysis model is fine-tuning its exploration strategy over time, focusing on areas that had not been explored adequately in previous iterations. This phenomenon likely arises from the adjustment of fuzzing parameters or the identification of under-represented code regions that require specific, less obvious inputs to trigger new transitions. It is also possible that, during the later stages of fuzzing, the model is able to exploit specific patterns learned from previous inputs, resulting in more targeted exploration that unveils hidden code paths.

The behaviour observed in the sensitivity analysis model suggests that a conditional branch (e.g., an "if" statement) was executed differently towards the end of the iterations, resulting in the discovery of a new code path. This shift in execution flow is likely due to the model's ability to discover specific input values that alter the conditions of these branches, causing them to take different paths that were previously unexecuted. As the fuzzing model gains more insight into the program's control flow through sensitivity analysis, it becomes better at identifying the right conditions to trigger these unvisited branches, leading to the discovery of new edges.

Although sensitivity analysis did not outperform the LReLU model in edge coverage, even with the distinct behaviour of new edge discovery, the outcome remains significant and provides meaningful insights for our research findings. The distinctive performance of sensitivity analysis, particularly its ability to uncover new edges in later iterations, highlights its potential as a complementary approach to traditional models like LReLU. While the LReLU model may cover a broad spectrum of edges more quickly, sensitivity analysis provides a more strategic exploration that can uncover additional code paths in later stages of fuzzing, offering a deeper understanding of program behaviour.

5.5. Discovery of unique bugs

The baseline model demonstrates significantly lower bug detection across the evaluated tools, except for two targets: `size` and `readelf`, where LReLU and its variant incorporating Sensitivity analysis still rank closely behind. Across other targets, LReLU and its combined variants exhibit a strong capability for detecting unique bugs, achieving top ranks as follows: LReLU in its pure form identified 591 unique bugs

in the djpeg target, while its combination with Sensitivity analysis detected 154 unique bugs in `mutool`. Additionally, when integrated with Nadam, LReLU identified 22 unique bugs in the `nm` target. Notably, LReLU, in conjunction with the baseline model, also outperformed other models on the `size` target. Meanwhile, Nadam consistently achieved a first-place ranking for `mutool`, a second-place ranking for `djpeg`, and a third-place ranking for `size`, following LReLU combined with Sensitivity analysis.

In summary, LReLU demonstrates a positive impact on the identification of unique bugs, with Nadam also contributing significantly in the our findings.

5.6. Total executions and max dimension

In this section, we examine both Total Executions and Maximum Dimension. In terms of total executions, we found that a higher total number of executions can sometimes correlate with increased iterations and improved coverage; however, efficiency varies among different models. For instance, the Sensitivity model demonstrates effective bug detection with fewer total executions in specific tools, such as `mutool` and `nm`. Additionally, regarding data dimension, LReLU, Nadam, and their hybrid model consistently yield reliable results in enhancing coverage dimensions across nearly all targets. In contrast, the Sensitivity analysis and its variant achieve commendable but lower rankings across only four targets (i.e., `djpeg`, `mutool`, `nm`, and `readelf`), while still outperforming the baseline model overall.

5.7. Analysing the efficacy of lrelu and its variants compared to the baseline through the lens of game theory

The incorporation of LReLU activation layers significantly enhances fuzzing performance relative to the baseline ReLU function employed in MTFuzz, as well as in comparison to other recommended techniques discussed in this paper, such as sensitivity analysis. Models that utilise LReLU and its variant integrated with sensitivity analysis consistently achieve higher overall maximum accuracy, reduced loss rates, and improved edge coverage across a range of evaluation tools. This suggests that LReLU is more effective in capturing nuanced patterns within the data, thereby leading to more successful fuzzing outcomes.

Furthermore, when evaluating the Nadam optimisation technique against the baseline Adam optimiser, Nadam exhibits a pronounced capacity to enhance edge coverage and unique bug detection in certain scenarios, which should not be overlooked. Notably, the performance of Nadam is comparable to that of hybrid models, highlighting its potential as a robust alternative for optimising fuzzing performance. Additionally, the implementation of post-training sensitivity analysis significantly influences the overall effectiveness of fuzzing outcomes. By fine-tuning model parameters and enhancing decision-making processes, sensitivity analysis not only elevates accuracy but also substantially contributes to the efficiency of performance in some targets.

The performance of LReLU and its variants, including those with sensitivity analysis and Nadam, can indeed be interpreted through the lens of game theory, particularly in terms of strategy selection and equilibrium concepts.

In this framework, various activation functions and optimisation techniques, such as LReLU, sensitivity with LReLU, and sensitivity alone, can be conceptualised as competing strategies within a game-theoretic context. Each strategy is designed to optimise a particular outcome as"payoff". The "payoff" in this scenario can be interpreted as the model's performance metrics, such as accuracy, loss, edge coverage, and unique bug detection. LReLU consistently achieves the highest performance across all metrics, making it the dominant strategy, as it yields the greatest payoff compared to the other strategies.

The performance hierarchy, based on maximum accuracy, indicates a potential equilibrium in the model prediction game, with LReLU emerging as the top-performing strategy. This is followed by the combination of sensitivity and LReLU, which represents an equilibrium state for these methods. Lastly, sensitivity alone performs the weakest across nearly all targets among them. These techniques consistently outperform Nadam and its variant, which rank lower across all targets in terms of maximum accuracy. This indicates that methods such as LReLU and its variant with sensitivity are more effective across almost all target metrics, yielding the highest payoff as defined by Eq. (27). This can be mathematically expressed by comparing the partial derivatives of the payoff function $\mathcal{U}$ with respect to λ, ν, and σ. If the value of λ consistently yields a higher payoff than the other strategies, LReLU can be considered a dominant strategy. Thus, based on our experimental findings, the following order represents the dominance strategy:

$$\frac{\partial \mathcal{U}}{\partial \lambda} > \frac{\partial \mathcal{U}}{\partial \sigma} > \frac{\partial \mathcal{U}}{\partial \nu} \tag{28}$$

This order suggests that once the models are trained using these techniques, they demonstrate varying levels of accuracy and reach a state of equilibrium within their variants, where transitioning to an alternative technique would not lead to significant improvements in results for the specific problem at hand and it aligns closely with the principles of adaptive strategies in game theory. It is because when considering the iterative training process of the models, it can be viewed as a dynamic game where strategies evolve in response to feedback (performance metrics) over time. The modifications facilitated by sensitivity analysis represent a form of strategic adaptation, enabling the model to enhance its performance in reaction to the results observed during the training phase.

In conclusion, analysing LReLU through game theory emphasises the strategic selection of methods and highlights the dynamics of optimal performance. While the baseline model is functional, there is considerable potential for improvement.

6. Future work

This study provides valuable insights into the integration of LReLU, Nadam, and sensitivity analysis for improving fuzzing techniques. However, our use of fixed hyperparameters – LReLU with an α of 0.01 and Nadam with an η of 0.001 – limits the scope of exploration. The impact of varying α and η values on performance remains underexplored, highlighting a key avenue for future research. Systematic hyperparameter tuning, using methods like grid search, Bayesian optimisation, or reinforcement learning, could uncover a broader range of values and enhance both model performance and bug detection.

Furthermore, exploring alternative activation functions, such as Parametric Rectified Linear Unit (PReLU) [55], Randomised ReLU (RReLU), Gaussian Error Linear Units (GELUs), and Self-Normalising Neural Networks [56], is essential. These functions mitigate the issue of dead neurons in traditional ReLU-based activations by providing non-zero gradients for negative inputs. Notably, PReLU and GELUs have shown promise in enhancing model performance by improving gradient flow, which is critical for fuzzing tasks requiring extensive path exploration. These activation functions may enable more efficient gradient propagation, uncovering previously unexplored paths. A comparative analysis of their effects on gradient flow, network generalisation, and computational efficiency could provide valuable insights into their potential for advancing fuzzing techniques and addressing challenges related to gradient propagation and model convergence.

Furthermore, Our study encountered hardware limitations that restricted the expansion of testing to additional scenarios, such as the use of LSTM networks or Transformer models. LSTMs are effective at handling sequential data and maintaining context over time, while Transformers, with their self-attention mechanisms, excel in capturing long-range dependencies. These features could enhance fuzzing tasks, particularly when dealing with time-dependent or sequence-sensitive inputs, enabling better coverage and more efficient exploration of fuzzing paths. Integrating LSTM or Transformer models may offer

a promising approach to improving fuzzing architectures by better contextualising temporal inputs and identifying greater path diversity.

Future work should aim to overcome these constraints, enabling the evaluation of a broader range of targets and configurations. This would provide valuable insights into the applicability and robustness of our findings across diverse contexts. By addressing these areas, future research could enhance the understanding and implementation of effective fuzzing methodologies, particularly regarding activation layers, optimisers, and sensitivity analysis. Ultimately, such efforts would contribute to more robust bug detection and improve software testing practices.

7. Conclusion

This paper investigates the integration of LReLU, Nadam, and sensitivity analysis to enhance fuzzing for accuracy rate, loss rate, edge coverage, and bug detection. Our findings reveal that LReLU, especially when combined with sensitivity analysis, significantly enhances detection efficiency, with performance improvements varying depending on the targeted bugs.

LReLU enhances classification and regression performance by preventing neuron inactivity and allowing small negative activations to contribute to feature extraction. This improves edge/context sensitivity, making LReLU a superior alternative to ReLU. In classification, it boosts gradient propagation and stability, while in regression, it mitigates vanishing gradients, resulting in smoother loss landscapes and better convergence. When paired with sensitivity analysis, LReLU ensures consistent gradient flow across both positive and negative regions, optimising learning by focusing on layers with higher sensitivity. This approach strengthens model resilience, especially against adversarial challenges.

Nadam demonstrates rapid convergence, particularly in adversarial sample generation tasks. However, its momentum update strategy can occasionally lead to fluctuations, which may limit its effectiveness for certain targets. While Nadam is an excellent choice for tasks that demand fast convergence, Adam remains a dependable alternative for applications where stable and predictable performance is essential.

The adaptive learning stage, guided by sensitivity analysis, prioritised fuzzing test cases based on the magnitudes of their gradients. By using backpropagation to minimise a multi-task loss function, task weights were dynamically adjusted according to their relative importance, enhancing the efficiency and effectiveness of the fuzz testing process. This framework offers valuable insights into the strategic selection of methods, emphasising the competitive dynamics that impact model performance. Understanding these dynamics will enable future research to better adapt the proposed techniques and optimise outcomes.

This study highlights significant advancements in fuzzing techniques while acknowledging the limitations of these approaches. The performance of LReLU is sensitive to hyperparameter choices, necessitating further tuning to ensure consistent results across tasks. We recommend exploring hyperparameter optimisation strategies, such as grid search or Bayesian optimisation, to enhance LReLU's stability. Although Nadam accelerates convergence, its susceptibility to fluctuations may limit its applicability in certain contexts. Future research could focus on developing hybrid models that combine Nadam with other optimisers to address momentum-related fluctuations or refining its momentum update strategy for improved stability across tasks.

Ultimately, this research provides a strategic framework for improving bug detection and software testing, highlighting the importance of ongoing exploration in these methodologies. Future work should focus on overcoming existing limitations and refining these techniques to boost their performance across a wider array of tasks, ensuring their effectiveness in real-world applications.

CRediT authorship contribution statement

Sadegh Bamohabbat Chafjiri: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Phil Legg:** Writing – review & editing, Validation, Supervision, Methodology, Conceptualization. **Michail-Antisthenis Tsompanas:** Writing – review & editing, Supervision. **Jun Hong:** Writing – review & editing, Validation, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This research was supported by the PhD Studentship scheme within College of Arts, Technology and Engineering at the University of the West of England.

Data availability

Data will be made available on request.

References

[1] B.P. Miller, L. Fredriksen, B. So, An empirical study of the reliability of UNIX utilities, Commun. ACM 33 (12) (1990) 32–44, http://dx.doi.org/10.1145/96267.96279.

[2] M. Woo, S.K. Cha, S. Gottlieb, D. Brumley, Scheduling black-box mutational fuzzing, in: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, CCS '13, Association for Computing Machinery, New York, NY, USA, 2013, pp. 511–522, http://dx.doi.org/10.1145/2508859.2516736.

[3] Peach fuzzer, 2023, https://peachfuzzer.com/. (last accessed: 27 May 2023).

[4] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, G. Vigna, SNOOZE: Toward a stateful NetwOrk protocol fuzZEr, in: Proceedings of the Information Security Conference, 2006.

[5] P. Godefroid, A. Kiezun, 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.

[6] V.-T. Pham, M. Böhme, A. Roychoudhury, Model-based whitebox fuzzing for program binaries, in: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, 2016, pp. 543–553.

[7] R. Hastings, B. Joyce, Purify: Fast detection of memory leaks and access errors, in: Proceedings of USENIX Winter'92 Conference, 1992, pp. 125–138.

[8] M. Felderer, M. Büchler, M. Johns, A.D. Brucker, R. Breu, A. Pretschner, Chapter one - security testing: A survey, in: A. Memon (Ed.), in: Advances in Computers, vol. 101, Elsevier, 2016, pp. 1–51, http://dx.doi.org/10.1016/bs.adcom.2015.11.003.

[9] C. Miller, Z.N.J. Peterson, Analysis of Mutation and Generation-Based Fuzzing, Tech. Rep., 2007.

[10] X. Zhu, S. Wen, S. Camtepe, Y. Xiang, Fuzzing: A survey for roadmap, ACM Comput. Surv. 54 (11s) (2022) 1–36, http://dx.doi.org/10.1145/3512345.

[11] R. Majumdar, K. Sen, Hybrid concolic testing, in: Proceedings of the 29th International Conference on Software Engineering, 2007, pp. 416–426, http://dx.doi.org/10.1109/ICSE.2007.41.

[12] I. Yun, S. Lee, M. Xu, Y. Jang, T. Kim, QSYM : A practical concolic execution engine tailored for hybrid fuzzing, in: 27th USENIX Security Symposium, USENIX Security 18, 2018, pp. 745–761.

[13] D. Molnar, P. Godefroid, M. Levin, Automated whitebox fuzz testing, in: Network and Distributed System Security Symposium, NDSS, 2008, pp. 416–426.

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

[15] S. Mallissery, Y.-S. Wu, Demystify the fuzzing methods: A comprehensive survey, ACM Comput. Surv. 56 (3) (2023) http://dx.doi.org/10.1145/3623375.

[16] G.J. Saavedra, K.N. Rodhouse, D.M. Dunlavy, P.W. Kegelmeyer, A review of machine learning applications in fuzzing, 2019, http://dx.doi.org/10.48550/ARXIV.1906.11133, arXiv, arXiv:1906.11133.

[17] Y. Wang, P. Jia, L. Liu, J. Liu, A systematic review of fuzzing based on machine learning techniques, PLoS One 15 (2020) http://dx.doi.org/10.1371/journal.pone.0237749.
[18] S. Miao, J. Wang, C. Zhang, Z. Lin, J. Gong, X. Zhang, J. Li, Deep learning in fuzzing: A literature survey, in: 2022 IEEE 2nd International Conference on Electronic Technology, Communication and Information, ICETCI, 2022, pp. 220–223, http://dx.doi.org/10.1109/ICETCI55101.2022.9832143.
[19] C. Daniele, S.B. Andarzian, E. Poll, Fuzzers for stateful systems: Survey and research directions, ACM Comput. Surv. (2024) http://dx.doi.org/10.1145/3648468, Just Accepted.
[20] S. Bamohabbat Chafjiri, P. Legg, J. Hong, M.-A. Tsompanas, Vulnerability detection through machine learning-based fuzzing: A systematic review, Comput. Secur. 143 (2024) 103903, http://dx.doi.org/10.1016/j.cose.2024.103903, URL https://www.sciencedirect.com/science/article/pii/S0167404824002050.
[21] S. Tripathi, G. Grieco, S. Rawat, Exniffer: Learning to prioritize crashes by assessing the exploitability from memory dump, in: 2017 24th Asia-Pacific Software Engineering Conference, APSEC, 2017, pp. 239–248, http://dx.doi.org/10.1109/APSEC.2017.30.
[22] L. Zhang, V.L.L. Thing, Assisting vulnerability detection by prioritizing crashes with incremental learning, in: TENCON 2018 - 2018 IEEE Region 10 Conference, 2018, pp. 2080–2085, http://dx.doi.org/10.1109/TENCON.2018.8650188.
[23] Y. Chen, M. Ahmadi, R. Mirzazade farkhani, B. Wang, L. Lu, MEUZZ: Smart seed scheduling for hybrid fuzzing, in: Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses, RAID '20, 2020.
[24] Y.-D. Lin, Y.-K. Lai, Q.T. Bui, Y.-C. Lai, ReFSM: Reverse engineering from protocol packet traces to test generation by extended finite state machines, J. Netw. Comput. Appl. 171 (2020) 102819, http://dx.doi.org/10.1016/j.jnca.2020.102819, URL https://www.sciencedirect.com/science/article/pii/S1084804520302897.
[25] Y. Huang, H. Shu, F. Kang, Y. Guang, Protocol reverse-engineering methods and tools: A survey, Comput. Commun. 182 (2022) 238–254, http://dx.doi.org/10.1016/j.comcom.2021.11.009, URL https://www.sciencedirect.com/science/article/pii/S0140366421004382.
[26] V. Raychev, M. Vechev, A. Krause, Predicting program properties from "big code", in: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '15, Association for Computing Machinery, New York, NY, USA, 2015, pp. 111–124, http://dx.doi.org/10.1145/2676726.2677009.
[27] X. Sun, Y. Fu, Y. Dong, Z. Liu, Y. Zhang, Improving fitness function for language fuzzing with PCFG model, in: 2018 IEEE 42nd Annual Computer Software and Applications Conference, Vol. 01, COMPSAC, 2018, pp. 655–660, http://dx.doi.org/10.1109/COMPSAC.2018.00098.
[28] J. Wang, B. Chen, L. Wei, Y. Liu, Skyfire: Data-driven seed generation for fuzzing, in: 2017 IEEE Symposium on Security and Privacy, SP, 2017, pp. 579–594, http://dx.doi.org/10.1109/SP.2017.23.
[29] S. Karamcheti, G. Mann, D. Rosenberg, Adaptive grey-box fuzz-testing with Thompson sampling, in: Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security, AISec '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 37–47, http://dx.doi.org/10.1145/3270101.3270108.
[30] D. She, R. Krishna, L. Yan, S. Jana, B. Ray, MTFuzz: Fuzzing with a multi-task neural network, in: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, in: ESEC/FSE 2020, Association for Computing Machinery, New York, NY, USA, 2020, pp. 737–749, http://dx.doi.org/10.1145/3368089.3409723.
[31] A.L. Maas, A.Y. Hannun, A.Y. Ng, et al., Rectifier nonlinearities improve neural network acoustic models, in: Proc. Icml, vol. 30, Atlanta, GA, 2013, p. 3.
[32] T. Dozat, Incorporating nesterov momentum into adam, in: ICLR 2016 - Workshop International Conference on Learning Representations, Caribe Hilton, San Juan, Puerto Rico, 2016, URL http://www.iclr.cc.
[33] S. Ruder, An overview of gradient descent optimization algorithms, 2017, arXiv:1609.04747, URL https://arxiv.org/abs/1609.04747.
[34] J.Y. Choi, C.-H. Choi, Sensitivity analysis of multilayer perceptron with differentiable activation functions, IEEE Trans. Neural Netw. 3 (1) (1992) 101–107, http://dx.doi.org/10.1109/72.105422.
[35] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, S. Jana, NEUZZ: Efficient fuzzing with neural program smoothing, in: 2019 IEEE Symposium on Security and Privacy, Vol. 1, SP, 2019, pp. 803–817, http://dx.doi.org/10.1109/SP.2019.00052.
[36] P. Chen, H. Chen, Angora: Efficient fuzzing by principled search, in: 2018 IEEE Symposium on Security and Privacy, SP, 2018, pp. 711–725, http://dx.doi.org/10.1109/SP.2018.00046.
[37] C. Lemieux, K. Sen, FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 475–485, http://dx.doi.org/10.1145/3238147.3238176.
[38] M. Zalewski, American fuzzy lop (AFL), 2023, https://lcamtuf.coredump.cx/afl/. (last accessed: 27 May 2023).
[39] M. Böhme, V.-T. Pham, A. Roychoudhury, Coverage-based greybox fuzzing as Markov chain, in: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS '16, Association for Computing Machinery, New York, NY, USA, 2016, pp. 1032–1043, http://dx.doi.org/10.1145/2976749.2978428.
[40] M. Sutton, A. Greene, P. Amini, Fuzzing: Brute Force Vulnerability Discovery, Addison-Wesley Professional, 2007.
[41] A. Takanen, J.D. Demott, C. Miller, Fuzzing for Software Security Testing and Quality Assurance, second ed., Artech House, 2018.
[42] G. Grieco, G.L. Grinblat, L. Uzal, S. Rawat, J. Feist, L. Mounier, Toward large-scale vulnerability discovery using machine learning, in: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy, CODASPY '16, Association for Computing Machinery, New York, NY, USA, 2016, pp. 85–96, http://dx.doi.org/10.1145/2857705.2857720.
[43] Z. Hu, J. Shi, Y. Huang, J. Xiong, X. Bu, GANFuzz: A GAN-based industrial network protocol fuzzing framework, in: Proceedings of the 15th ACM International Conference on Computing Frontiers, CF '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 138–145, http://dx.doi.org/10.1145/3203217.3203241.
[44] C. Cummins, P. Petoumenos, A. Murray, H. Leather, Compiler fuzzing through deep learning, in: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, in: ISSTA 2018, Association for Computing Machinery, New York, NY, USA, 2018, pp. 95–105, http://dx.doi.org/10.1145/3213846.3213848.
[45] L.H. Park, S. Chung, J. Kim, T. Kwon, GradFuzz: Fuzzing deep neural networks with gradient vector coverage for adversarial examples, Neurocomputing 522 (2023) 165–180, http://dx.doi.org/10.1016/j.neucom.2022.12.019, URL https://www.sciencedirect.com/science/article/pii/S0925231222015168.
[46] K. Patil, A. Kanade, Greybox fuzzing as a contextual bandits problem, 2018, arXiv, arXiv:1806.03806.
[47] T. Li, X. Wan, M.M. Özbek, AgentFuzz: Fuzzing for deep reinforcement learning systems, in: 2022 IEEE International Symposium on Software Reliability Engineering Workshops, ISSREW, 2022, pp. 110–113, http://dx.doi.org/10.1109/ISSREW55968.2022.00049.
[48] M. Wu, L. Jiang, J. Xiang, Y. Zhang, G. Yang, H. Ma, S. Nie, S. Wu, H. Cui, L. Zhang, Evaluating and improving neural program-smoothing-based fuzzing, in: Proceedings of the 44th International Conference on Software Engineering, ICSE '22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 847–858, http://dx.doi.org/10.1145/3510003.3510089.
[49] N. Nichols, M. Raugas, R. Jasper, N. Hilliard, Faster fuzzing: Reinitialization with deep neural models, 2017, arXiv:1711.02807.
[50] Y. Li, S. Ji, C. Lyu, Y. Chen, J. Chen, Q. Gu, C. Wu, R. Beyah, V-Fuzz: Vulnerability prediction-assisted evolutionary fuzzing for binary programs, IEEE Trans. Cybern. 52 (5) (2022) 3745–3756, http://dx.doi.org/10.1109/TCYB.2020.3013675.
[51] S. Jeon, J. Moon, Dr.PathFinder: hybrid fuzzing with deep reinforcement concolic execution toward deeper path-first search, Neural Comput. Appl. 34 (13) (2022) 10731–10750, http://dx.doi.org/10.1007/s00521-022-07008-8.
[52] R. Krishna, Dongdongshe, Helpacksi, Fuzzing with a multi-task neural network: nn.py, 2020, GitHub, URL https://github.com/rahlk/MTFuzz/blob/master/nn.py. (Accessed: 04 October 2024), GitHub repository, https://github.com/rahlk/MTFuzz/blob/master/nn.py.
[53] J. Xu, Z. Li, B. Du, M. Zhang, J. Liu, Reluplex made more practical: Leaky ReLU, in: 2020 IEEE Symposium on Computers and Communications, ISCC, 2020, pp. 1–7, http://dx.doi.org/10.1109/ISCC50000.2020.9219587.
[54] S.R. Dubey, S.K. Singh, B.B. Chaudhuri, Activation functions in deep learning: A comprehensive survey and benchmark, Neurocomputing 503 (2022) 92–108, http://dx.doi.org/10.1016/j.neucom.2022.06.111, URL https://www.sciencedirect.com/science/article/pii/S0925231222008426.
[55] K. He, X. Zhang, S. Ren, J. Sun, Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification, in: 2015 IEEE International Conference on Computer Vision, ICCV, 2015, pp. 1026–1034, http://dx.doi.org/10.1109/ICCV.2015.123.
[56] Y. Bai, RELU-function and derived function review, SHS Web Conf. 144 (2022) 02006, http://dx.doi.org/10.1051/shsconf/202214402006.