Destabilizing Neurons to Generate Challenging Neural Network Verification Benchmarks

Linhan Li
Department of Computer Science
George Mason University
Fairfax, USA
lli34@gmu.edu

ThanhVu Nguyen
Department of Computer Science
George Mason University
Fairfax, USA
tvn@gmu.edu

***Abstract*—Neural Network Verification has made significant progress in recent years, with the development of numerous verification techniques and tools. However, the field still lacks high-quality benchmarks for systematically evaluating and improving these tools. As verification techniques advance, many existing benchmarks have become too trivial, while the harder ones often remain unsolvable. Several recent efforts have attempted to address this gap, typically by retraining or distilling neural networks to create new benchmarks. However, such approaches are computationally expensive and often produce benchmarks with unknown or unverifiable ground truth.**

In this paper, we introduce `ReluSplitter`, an automatic benchmark generation tool for DNN verifiers. `ReluSplitter` takes existing verification benchmarks as input and strategically destabilizes stable neurons to increase verification difficulty. This transformation is semantics-preserving by construction: every `ReluSplitter`-generated benchmark is guaranteed to have exactly the same ground truth as the original benchmark. This makes `ReluSplitter` particularly valuable for assessing verifier correctness and performance.

Our evaluation demonstrates that `ReluSplitter` can significantly increase the difficulty of existing benchmarks, effectively challenging state-of-the-art DNN verifiers. We believe `ReluSplitter` offers a practical and principled way to generate benchmarks with tunable difficulty and verifiable ground truth, contributing a much-needed resource for the neural network verification community.

***Index Terms*—Neural network Verification, Benchmark Generation.**

I. Introduction

The rapid development of Deep Neural Networks (DNNs) has fueled significant advancements in a wide range of applications, from autonomous systems to medical diagnostics [1]–[6]. However, as DNNs are increasingly deployed in safety-critical settings, the need for ensuring their correctness and robustness has become paramount. This has led to the emergence of DNN Verification (DNNV), a field dedicated to formally verifying the behavior of DNNs against specified properties. Over the past few years, the DNNV community has been fruitful, with numerous verification tools and techniques being developed to address the challenges of DNNV [7]–[14].

As in other fields of verification, the comparative evaluation of techniques and tools serves as a driving force in scaling their performance. The annual neural network verification competition (VNN-COMP) has been the main platform for evaluating and comparing DNN verifiers [7], [15]–[18]. However, the benchmarks used in VNN-COMP are often provided by verifier authors, leading to a lack of diversity and challenge in the benchmarks. In recent years, VNN-COMP has actively called for benchmarks that are *"not so hard that none of the instances can be solved by any participant, but also not so easy that every tool can solve all of them"* [15]

The importance of robust benchmarking has sparked growing interest in benchmark generation for DNN verification. Existing efforts to generate new benchmarks typically involve training networks with embedded properties [19] or modifying models through distillation or retraining [20], [21]. While these methods can create challenging instances, they are computationally expensive and require training data that may not always be available. Moreover, training and distillation often produce benchmarks with unknown or unverifiable ground truth, making evaluation difficult.

These challenges in current benchmark generation highlight the need for approaches that can more directly control benchmark difficulty and ensure verifiable ground truth. To develop such methods, it's crucial to understand the fundamental factors contributing to the complexity of DNN verification. It is generally considered that the difficulty of DNNV largely stems from the non-linearities introduced by activation functions [22]–[24]. Among these, the Rectified Linear Unit (ReLU) is particularly important, as its piecewise linear nature gives rise to the notion of neuron stability.

Neuron stability has been well studied in the literature. Prior works have shown it is closely related to the complexity of verifying neural networks. For example, [9], [25] leverage neuron stability to assist verification, while [26]–[28] incorporate it at training time to maximize the verifiability of the trained networks. However, to the best of our knowledge, no existing work has leveraged neuron stability for the purpose of benchmark generation.

a) Our work: We present `ReluSplitter`, a technique and tool for automatically synthesizing challenging verification benchmarks by *destabilizing stable neurons*. Unlike prior approaches, `ReluSplitter` is both efficient and semantically exact: it preserves the behavior of the original model and eliminates the need for training or distillation. Given a network and property, `ReluSplitter` identifies and destabilizes stable

neurons in the network to increase verification complexity. This process forces verifiers to perform additional branching and abstraction, thereby helping to determine their scalability and test the effectiveness of their underlying heuristics.

We evaluate `ReluSplitter` using VNN-COMP benchmarks and leading verifiers, demonstrating its effectiveness in producing more challenging yet valid benchmarks. On the `MNIST_FC` benchmark, applying `ReluSplitter` results in a consistent average slowdown of over 5x. Moreover, on the convolutional and residual benchmarks `Oval 21` and `SRI_ResNet`, it transforms previously simple instances (solvable in under 120s) into intractable ones that remained unsolvable even after a 1200-second timeout.

We believe `ReluSplitter` provides a practical and principled approach to improving benchmark diversity and rigor in the DNN verification community.

b) Contributions: We make the following contributions:

- We propose `ReluSplitter`, a technique and tool for DNNV benchmark generation that synthesizes new benchmark instances from existing ones. To the best of our knowledge, `ReluSplitter` is the first benchmark generation approach that both preserves ground truth and leverages neuron stability to increase verification difficulty.
- We provide an easy-to-use, configurable, and extensible open-source implementation of `ReluSplitter`, enabling researchers to generate custom benchmarks for evaluating verification techniques and to support future research and optimization.
- We conduct an extensive evaluation using four widely-used state-of-the-art (SOTA) verifiers and four ReLU-based verification benchmarks from VNN-COMP, covering three DNN architectures: fully connected networks, convolutional networks, and ResNet.

II. Background

A. DNN Verification (DNNV)

Given a DNN N and a property ϕ, the DNNV problem asks if ϕ holds on N. Typically, ϕ has the form $\phi_{in} \Rightarrow \phi_{out}$, where ϕ_{in} and ϕ_{out} are sets of linear inequalities over the input and outputs of N, respectively. A DNN verifier attempts to find an input to N that satisfies ϕ_{in} but violates ϕ_{out} (i.e. a counterexample). If no such input exists, ϕ is a valid property of N (*UNSAT*). Otherwise, ϕ is not valid (*SAT*) and a counterexample is returned.

The DNNV problem is NP-complete for networks using the popular ReLU activation function [22]. Modern DNN verifiers often employ the Branch and Bound (BnB) approach, which splits (or *branches*) the problem into smaller subproblems, and applies abstraction to compute the *bounds* of neuron values [10], [29]–[35]. Some recent verifiers leverage the concept of neuron stability to reduce branching and abstraction [9], [27], [36].

B. DNNV Benchmarks

A DNNV benchmark consists of a set of DNNV problem instances. Benchmarks are the primary means to evaluate the performance of DNN verifiers. *First-party* benchmarks are created by a DNN verifier developer when evaluating their tool, e.g., the well-known ACAS Xu benchmark [22]. Such benchmarks are typically tested with a limited set of competitors beyond the developer. *Third-party* benchmarks, in contrast, are created independently of any single verifier, often with the goal of broadly assessing verifier performance across tools. These include not only benchmarks designed purely for comparative evaluation—such as those in the VNN-COMP [15]–[18]—but also problem sets created by researchers applying DNN verification to new domains or properties. For example, a third-party benchmark might arise when verifying system specifications in power systems [18]. Such benchmarks tend to have a larger and more diverse set of problems. VNN-COMPs include a number of first-party benchmarks that are increasingly becoming too easy for verifiers to solve, and thus actively encourage the creation of more third-party benchmarks—the purpose of `ReluSplitter`.

C. Neuron Stability

The ReLU (Rectified Linear Unit) activation function is defined as $\text{ReLU}(z) = \max(0, z)$, where z is the pre-activation value produced by a neuron. A ReLU neuron is *active* if z is positive, in which case the output equals the pre-activation: $\text{ReLU}(z) = z$. Otherwise, the neuron is *inactive* if $z \leq 0$, and its output is zero: $\text{ReLU}(z) = 0$.

A ReLU neuron is said to be *stable* over a given input interval if it remains entirely in one of these two states—either always active or always inactive—throughout the interval. Let $z(x)$ and $n(x)$ be the neuron's pre-activation and post-activation values on input x, respectively. Formally, the ReLU neuron is stable over an interval $[l, u]$ if either:

$$\forall x \in [l, u]: \quad n(x) = z(x) \quad \text{(always active)}$$

or

$$\forall x \in [l, u]: \quad n(x) = 0 \quad \text{(always inactive)}.$$

If the bounds on $z(x)$ straddle zero—i.e., $a \leq z(x) \leq b$ with $a < 0 < b$—then the neuron is considered *unstable*, as it may switch between activation states within the interval.

III. Overview

A. Overview

Fig. 1 gives an overview of `ReluSplitter`, which takes a given DNNV instance (a DNN and property pair) and converts it into a more challenging yet semantically equivalent version by *destabilizing stable neurons*.

a) Pre-processing: `ReluSplitter` begins with a *pre-processing* phase, where `ReluSplitter` identifies layers that can be split (e.g., fully connected or convolutional layers with ReLU activation) and performs *neuron bound analysis* (e.g., using interval abstraction) to obtain a set of bounds that will be used to guide the process. Specifically, the bounds are

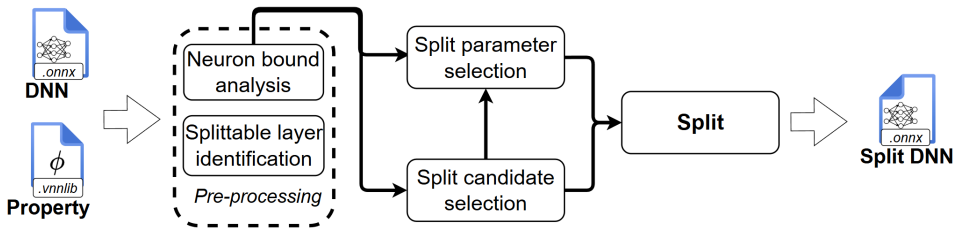
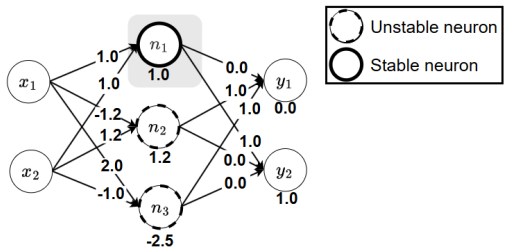


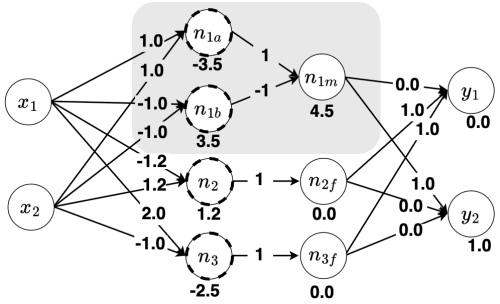
Fig. 1: `ReluSplitter` overview



(a) Input DNN N

$$(1 \leq x_1 \leq 4 \wedge 1 \leq x_2 \leq 2) \Rightarrow (y_1 \leq y_2)$$

(b) Property ϕ associated with the networks.



(c) Split DNN N'

Fig. 2: Illustration of `ReluSplitter` converting N to N'

calculated by propagating the input bound from the property (i.e. ϕ_{in}) through the network.

b) Split candidate selection: Then `ReluSplitter` checks the bounds and *selects* neurons for splitting. In `ReluSplitter`, we have implemented a few strategies for selecting the split candidates (e.g., stable neurons, unstable neurons, convolution filter with more stable outputs, etc.) and the user can also extend `ReluSplitter` with their own strategy. By default, it selects *stably active neurons*, which are usually more relevant to the property being verified (than stably inactive neurons) but do not cause branching and therefore make the DNNV instance easy for the verifier to solve.

c) Split parameter selection: After the split candidates are selected, `ReluSplitter` uses the bounds from earlier stage to decide split parameters for each split candidate. These parameters are crucial for constructing the split gadget, which takes place in the next step. There are three parameters to be decided, the scaling parameters s_{neg} and s_{pos}, and the split offset parameter τ. While the two scaling parameters can be decided without the bound information, deciding the split offset requires knowing the bound of the candidate.

d) Split: Finally, `ReluSplitter` uses the chosen parameters to split the candidates. This involves replacing the split candidates with *split gadgets*. The split gadget consists of two forward-facing neurons that introduce instability and a backward-facing neuron that ensures the gadget's output remains functionally identical to the neuron being replaced. `ReluSplitter` also applies any additional transformations needed to preserve the overall semantics of the original network. The result is a *split DNN* that, when verified against the same *property* ϕ, yields the same verification outcome (e.g., UNSAT) but poses a greater challenge for verifiers due to an increased number of unstable neurons.

B. Illustration

We illustrate `ReluSplitter` using the ReLU-based feed-forward network N (Fig. 2a) and the property ϕ (Fig. 2b) to derive the network N' (Fig. 2c). In these diagrams, the input variables are denoted by x_i, the hidden neurons by n_i, and the output neurons by y_i . Each edge is annotated with its weight, and each neuron shows its bias directly beneath it. For example, the original network N consists of two inputs x_1 and x_2, one fully connected layer with three neurons n_1, n_2, and n_3 , and two outputs y_1 and y_2 . The output of neuron n_1 is computed as $n_1 = ReLU(x_1 + x_2 + 1)$. In the transformed network N' , edges with a weight of zero are omitted for clarity.

a) Pre-processing: Since there is only a single fully connected layer in the network N, and it uses ReLU, `ReluSplitter` picks this layer to perform the split. `ReluSplitter` first extracts the input bounds from the pre-condition ϕ_{in}:

$$1 \leq x_1 \leq 4 \wedge 1 \leq x_2 \leq 2$$

`ReluSplitter` then uses *interval abstraction* [37] to bound the pre-activation value z_i of each neuron n_i in the layer.

$$3 \leq z_1 \leq 7, \quad -2.4 \leq z_2 \leq 2.4, \quad -2.5 \leq z_3 \leq 4.5$$

b) Split candidate selection: `ReluSplitter` selects split candidates using bound information. In this example, n_1 is classified as *stable* since the bound of z_1 is strictly positive,

while n_2 and n_3 are classified as *unstable* because their pre-activation bounds cross zero. Therefore, in this case, n_1 is the only neuron selected as a split candidate.

c) Split parameter selection: `ReluSplitter` proceeds to decide split parameters for the split candidate n_1. It first decides a pair of scaling parameters s_{neg}, s_{pos} to be used for the split. These parameters are multipliers that scale the weights and biases of the forward-facing neurons. Such scaling can cause distortion in some abstract domains, making them less accurate. It is necessary for the scaling parameters to have opposite signs. For simplicity, in this example we use $(s_{neg}, s_{pos}) = (-1, 1)$ to create a symmetric and straightforward split. It then decides the split offset τ. To make the resulting gadget unstable, it is necessary to pick a τ within the computed pre-activation bound of the split candidate. For example, we use $4.5 \in (3, 7)$ as the split offset for n_1.

d) Split: Finally, `ReluSplitter` performs the split on the candidates using the chosen parameters. In the example, n_1 was replaced by the split gadget consisting of two forward-facing split neurons n_{1a} and n_{1b}, and one backward-facing merge neuron n_{1m}, as defined below:

$$n_{1a} = ReLU(x_1 + x_2 - 3.5)$$
$$n_{1b} = ReLU(-x_1 - x_2 + 3.5)$$
$$n_{1m} = ReLU(n_{1a} - n_{1b} + 4.5)$$

The gadget is also highlighted by the gray rectangle in Fig. 2c. We can verify the instability of n_{1a} and n_{1b} by checking their bounds. Using the same *interval abstraction* as before, we have the following bounds:

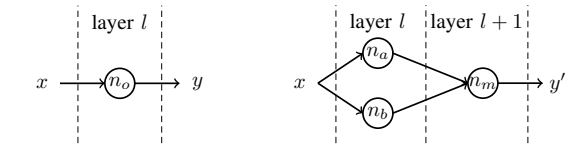
$$-1.5 \leq z_{1a} \leq 2.5, \quad -2.5 \leq z_{1b} \leq 1.5$$

Since both bounds cross zero, n_{1a} and n_{1b} are indeed unstable.

For neurons that are not split, `ReluSplitter` adds forwarding neurons to maintain the original behavior (e.g., n_{2f} forwards n_2 and n_{3f} forwards n_3). As a result, the transformation creates a network N' that has more unstable neurons than the original network N (4 vs. 2). This increases the number of possible branches in verification from 2^2 before splitting to 2^4 after splitting. Importantly, the verification ground truth is preserved because the gadgets do not alter the network's semantics, meaning N and N' are functionally equivalent. In this example, since the considered property is valid in N, it is also valid in N'.

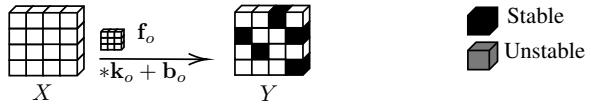
IV. The ReluSplitter Approach

The main idea of our `ReluSplitter` approach is to inject instability into neural networks by replacing original components (neurons or filters) with *split gadgets*. While these gadgets are designed to be unstable, they are semantically equivalent to the original components, ensuring the network's overall behavior remains unchanged. First, we describe the split gadgets for fully connected (FC) and convolutional (Conv) layers. Then, we introduce the *forwarding gadget*, which helps preserve network structure. Finally, we provide proof sketches for the gadgets' semantic equivalence.

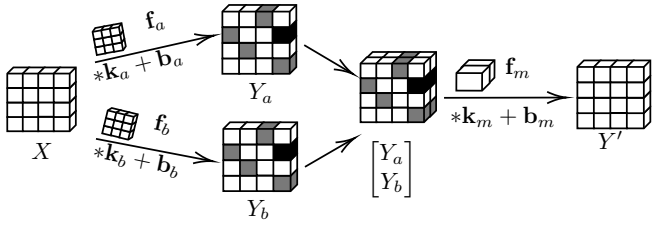


(a) Original Neuron n_o (b) n_o after the split

Fig. 3: Split Gadget for FC



(a) Original filter $\mathbf{f}_o$



(b) $\mathbf{f}_o$ after the split

Fig. 4: Split Gadget for Conv

Throughout this section, we use the term *component* to refer generically to a neuron in fully connected layers or a filter in convolutional layers.

A. Split Gadgets

The heart of `ReluSplitter` is the split gadgets, which replace stable neurons with semantically equivalent unstable multi-neuron gadgets. Each gadget has two scaling parameters denoted by s_{neg} and s_{pos}, and a split offset parameter τ. Together, these three parameters allow injection of instability at any neuron over any interval without altering the network's behavior.

The components are destabilized using the split offset τ, which must be chosen within the original component's pre-activation bounds (l, u). When this offset is subtracted from the forward-facing neurons, their bounds are shifted accordingly, resulting in the new bounds $(l - \tau, u - \tau)$. Because τ lies strictly between l and u, the shifted bounds are guaranteed to span zero—i.e., $l - \tau < 0 < u - \tau$. This ensures that the resulting components are unstable by construction.

Finally, the backward-facing component combines the outputs of the forward-facing components and cancels out the scaling and offset applied, restoring the original output.

Each splitting adds overhead in component count and depth. Splitting one neuron introduces two forward-facing components in the original layer and one backward-facing component in the next layer, effectively increasing depth. However, when multiple neurons are split from the same layer, their backward-facing components can be grouped into a single shared layer.

For instance, splitting 10 neurons from the same layer adds one extra layer, not 10.

The current implementation of `ReluSplitter` supports two types of popular layers: FC layers and Conv layers.

a) FC Layers: In an FC layer, neurons have individual weighted sums and do not share weights, thus we can introduce instability at individual neurons. As shown in Fig. 3, a stable neuron is replaced by a *three-neuron* gadget: (i) **two forward-facing neurons** (n_a, n_b) and (ii) **a backward-facing neuron** (n_m). As discussed, the two forward-facing neurons will be unstable, and the output of the backward-facing neuron is guaranteed to match the output of the original neuron n_o exactly. Denoting the weight and bias of neuron n_i as w_i and b_i, respectively, and the input of the neuron as x, the FC gadget is defined as follows:

$$n_a(x) = ReLU(\underbrace{s_{pos}w_o}_{w_a} \cdot x + \underbrace{s_{pos}(b_o - \tau)}_{b_a})$$
$$n_b(x) = ReLU(\underbrace{s_{neg}w_o}_{w_b} \cdot x + \underbrace{s_{neg}(b_o - \tau)}_{b_b})$$
$$n_m(x) = ReLU(\underbrace{[\frac{1}{s_{pos}}, \frac{1}{s_{neg}}]}_{w_m} \cdot \begin{bmatrix} n_a(x) \\ n_b(x) \end{bmatrix} + \underbrace{\tau}_{b_m})$$
where
$$s_{neg} < 0, s_{pos} > 0$$

b) Conv Layer: The gadget for Conv layers follows the same idea and generalizes the FC gadget. However, unlike FC, Conv layers apply spatially shared filters across the input tensor. Thus, instead of replacing a single neuron, we replace the entire convolutional filter to maintain spatial consistency. The split gadget for Conv layers, shown in Fig. 4, consists of (i) **two forward-facing filters** $(\mathbf{f}_a, \mathbf{f}_b)$, which produce destabilized output channels, and (ii) **one backward-facing filter** $(\mathbf{f}_m)$, which combines the destabilized channels to reconstruct the original output channel. The Conv gadget has the same parameters as the FC gadget, however, the selection of τ is slightly different because the bias of a filter is shared by all outputs in the channel. Because the split offset τ will be shared among all the outputs in the channel, it is unlikely that there is a single offset that can destabilize all outputs in the channel. To maximize the number of destabilized outputs, we take the bounds of each individual output into account, and use a sliding window to find the optimal τ value that destabilizes the most outputs. Let the $\mathbf{k}_i$, $\mathbf{b}_i$ be the kernel and bias of filter $\mathbf{f}_i$, the Conv gadget is given by ($*$: convolution operation):

$$\mathbf{f}_a(X) = ReLU(\underbrace{s_{pos}\mathbf{k}_o}_{\mathbf{k}_a} * X + \underbrace{s_{pos}(\mathbf{b}_o - \tau)}_{\mathbf{b}_a})$$
$$\mathbf{f}_b(X) = ReLU(\underbrace{s_{neg}\mathbf{k}_o}_{\mathbf{k}_b} * X + \underbrace{s_{neg}(\mathbf{b}_o - \tau)}_{\mathbf{b}_b})$$
$$\mathbf{f}_m(X) = ReLU(\underbrace{[[\frac{1}{s_{pos}}, \frac{1}{s_{neg}}]]}_{\mathbf{k}_m} * \begin{bmatrix} \mathbf{f}_a(X) \\ \mathbf{f}_b(X) \end{bmatrix} + \underbrace{\tau}_{\mathbf{b}_m})$$
where
$$s_{neg} < 0, s_{pos} > 0$$

B. Forwarding Gadget

The forwarding gadget, as the name suggests, passes the output of a component through without modification. It is introduced to bridge the structural mismatch between split gadgets—which span two layers—and normal (unsplit) components, which occupy only one. The forwarding gadget allows the output from the unsplit components to pass through an extra layer, thus ensuring a consistent layer structure.

The forwarding gadget can forward either a single output from a neuron in an FC layer or an entire output channel from a filter in a Conv layer. In the FC case, the gadget is a single neuron (placed in the subsequent layer) with its weights and biases set to zero, except for a single weight of 1 corresponding to the neuron being forwarded. In the Conv case, it is a 1x1 convolution with all weights zero except for a single weight of 1 on the forwarded channel. In both cases, the forwarding gadget performs a simple identity operation, preserving the original output while ensuring structural compatibility with the split gadgets.

C. The `ReluSplitter` algorithm

Algorithm 1: SplitModel

Input:
① A neural network $N = (l_1, l_2, \ldots, l_n)$ s.t. $N(x) = l_n(l_{n-1}(\ldots l_2(l_1(x))\ldots))$
② A property $\phi = (\phi_{\text{in}} \implies \phi_{\text{out}})$
③ A layer $l_i \in N$ to be split

Output: Another neural network N'

if l_i *is a FC/Conv layer with ReLU* **then**
  $bounds \leftarrow ComputeBound(N, \phi_{in}, l_i)$
  $l_{split}, l_{merge} \leftarrow SplitLayer(l_i, bounds)$
else
  Raise "layer l_i cannot be split"
return $N' = (l_1, \ldots, l_{i-1},\ l_{split},\ l_{merge}, l_{i+1}, \ldots, l_n)$

`ReluSplitter`'s main algorithm is described by Alg. 1. It takes as input a DNN $N = (l_1, l_2, \ldots, l_n)$, where the output is defined as $N(x) = l_n(l_{n-1}(\ldots l_2(l_1(x))\ldots))$; a property $\phi = (\phi_{in} \implies \phi_{out})$ to be verified; and a target ReLU layer $l_i \in N$ to be split.

Algorithm 2: SplitLayer

Input:

① A ReLU layer (FC or Conv) $l = [c_1, c_2, \ldots, c_k]$

where each component c_i is a weight-bias pair (w_i, b_i)

② The bounds for layer l, $bounds$

$\forall c \in l$, $bound[c]$ is the output bound(s) of c.

All bounds are given in the form (lb, ub), where lb is the lower bound and ub is the upper bound.

Output:

Two ReLU layers l_{split} and l_{merge}

s.t $l_{merge}(l_{split}(x)) = l(x)$

1 $select\_idxs \leftarrow SelectSplits(l, bounds)$
2 $l_{split}[k + len(select\_idxs)]$
3 $l_{merge}[k]$
4 $idx \leftarrow 1$
5 **for** $j \leftarrow 1$ **to** k **do**
6 **if** $j \in select\_idxs$ **then**
7 $s_{neg}, s_{pos} \leftarrow DecideScale(l[j], bounds[j])$
8 $\tau \leftarrow DecideOffset(l[j], bounds[j])$
9 $w, b \leftarrow l[j]$
10 $l_{split}[idx] \leftarrow (s_{pos}w,\ s_{pos}(b - \tau))$
11 $l_{split}[idx + 1] \leftarrow (s_{neg}(w),\ s_{neg}(b - \tau))$
12 $w_{merge} \leftarrow [0] * len(l_{split})$
13 $w_{merge}[idx], w_{merge}[idx + 1] \leftarrow \frac{1}{s_{pos}}, \frac{1}{s_{neg}}$
14 $l_{merge}[j] \leftarrow (w_{merge}, \tau)$
15 $idx \leftarrow idx + 2$
16 **else**
17 $l_{split}[idx] \leftarrow l[j]$
18 $w_{merge} \leftarrow [0] * len(l_{split})$
19 $w_{merge}[idx] \leftarrow 1$
20 $l_{merge}[j] \leftarrow (w_{merge}, 0)$
21 $idx \leftarrow idx + 1$
22 **return** l_{split}, l_{merge}

The algorithm requires l_i to be a fully connected (FC) or convolutional (Conv) layer with ReLU activation; if not, it aborts (line 5). This layer-wise approach is compatible with complex architectures like ResNets, as residual connections define the network's topology rather than constituting a distinct layer type that would be split.

For a valid l_i, `ReluSplitter` first computes pre-activation output bounds using the `ComputeBound` subroutine (line 2). It then calls `SplitLayer` to generate the two replacement layers, l_{split} and l_{merge} (line 3). Finally, it returns a new network, N', where l_i is replaced by its destabilized counterparts, l_{split} and l_{merge} (line 6).

a) SplitLayer: As shown in Alg. 2, this subroutine splits a ReLU layer $l = [c_1, c_2, \ldots, c_k]$, where each component c_i is a neuron (in FC layers) or a filter (in Conv layers), represented as a weight-bias pair (w_i, b_i). The function also receives a mapping $bounds$ giving pre-activation output bounds for each component in the form (lb, ub).

The algorithm begins by identifying split candidates using the `SelectSplits` subroutine (line 1). Two new layers are then initialized: l_{split} with k + `len(select_idxs)` components, and l_{merge} with k components (line 2–line 3) (k is the number of components in l). The index variable idx tracks the position in l_{split} during construction (line 4). Then, for each component j in layer l, the procedure branches based on whether j is selected for splitting. If j is not selected, it is directly copied into the split layer l_{split} (line 17), and a forwarding gadget is added to the merge layer l_{merge} to preserve functional equivalence (line 18–line 20). If j is selected for splitting, the parameters s_{neg}, s_{pos}, and τ are first determined using the `DecideScale` and `DecideOffset` subroutines (line 7–line 8). Based on these, two forward-facing components with scaled weights and shifted biases are added to l_{split} (line 10–line 11), and a backward-facing component is constructed in l_{merge} to recombine the split outputs into the original activation (line 12–line 14).

This construction ensures that $l_{merge}(l_{split}(x)) = l(x)$ for all inputs x. Finally, the replacement layers l_{split} and l_{merge} are returned.

b) ComputeBound (Alg. 1, line 2): This subroutine estimates pre-activation output bounds for a given layer using techniques such as Interval Bound Propagation (IBP) or zonotope abstractions. `ReluSplitter` leverages bounds computed using `auto-LiRPA` [38], a SOTA library for abstract interpretation on DNNs.

c) SelectSplits (Alg. 2, line 1): This subroutine identifies split candidates. By default, `ReluSplitter` selects stably active neurons in FC layers or filters in Conv layers with at least one stable output in their output channels. However, the selection strategy is configurable. For instance, users can opt to split unstable or stably active components, or define custom criteria.

d) DecideScale (Alg. 2, line 7): This subroutine determines the scaling parameters s_{pos} and s_{neg} used by the split. These values affect the magnitude of the weights and biases after splitting, which can distort some abstract domains, making them less tight. Currently, `ReluSplitter` does not have a systematic way of deciding them and thus only randomly picks values within a reasonable range that do not cause excessive floating point error when scaling the weight and bias.

e) DecideOffset (Alg. 2, line 8): This subroutine determines the split offset τ used in the split gadget. For FC neurons, the strategy depends on their stability: if the neuron is stable, τ is randomly selected within the neuron's pre-activation output bounds to intentionally destabilize it. For unstable neurons, the user can configure whether to bias τ toward the active or inactive region of the output range; these behaviors are controlled via command-line flags. For Conv filters, `ReluSplitter` chooses a τ value that destabilizes as many output locations as possible. It applies a sliding window algorithm to identify a range of τ values that intersect the

maximum number of spatial output bounds, and then selects a final value from within this range.

All of these—`DecideScale`, `DecideOffset`, `ComputeBound`, and `SelectSplits`—are modular and extensible. Users may customize or replace them to suit specific objectives or integration with external tools.

D. `ReluSplitter` is exact

Theorem 1 (`ReluSplitter`(FC) Gadget is Exact)**.** *For any neuron n_o with weight w_o, bias b_o, and post-ReLU output $n_o(x)$, the `ReluSplitter` split gadget (comprising neurons n_a, n_b, and n_m as defined in §IV, with any split parameters $s_{pos} > 0$, $s_{neg} < 0$, and offset τ) produces exactly the same output as $n_o(x)$ for all inputs x:*

$$n_m(x) = n_o(x) \quad \textit{for all } x.$$

Proof (Sketch). We prove this by showing the pre-activation output of the original neuron $z_o(x)$ is equal to $z_m(x)$ for all $x \in \mathbb{R}^d$. As both are ReLU neurons, $z_o(x) = z_m(x)$ inherently means $n_m(x) = n_o(x)$.

Since the computation of $z_m(x)$ involving evaluation of two ReLU neurons, n_a and n_b, there can be at most four possible activation patterns: only n_a active, only n_b active, both active, or both inactive. We consider each case separately.

In the first case, since n_a is active and n_b is inactive, $z_m(x) = \frac{1}{s_{pos}} z_a(x) + \tau$. Plug in $z_a(x) = s_{pos} w_o \cdot x + s_{pos}(b_o - \tau)$ and apply necessary algebraic simplification, we have $z_m(x) = w_o \cdot x + b_o$, which is exactly equal to z_o. The second case can be proved similarly.

In the third case, both n_a and n_b are active at the same time. However, we know this is impossible because they are the same affine function scaled by either s_{pos} or s_{neg}, and have opposing sign on any non-zero values.

The last case, can only occur when $z_a(x) = z_b(x) = 0$ as they always have opposing sign on non-zero values. It is easy to see $z_m(x) = \tau$ when both ReLU neuron are inactive. We can also infer $z_o(x) = \tau$ from $n_a(x) = n_b(x) = 0$. Thus, $z_m(x) = z_o(x)$ also holds for the last case.

Therefore, in all cases, the split gadget preserves the behavior of the original neuron. □

Theorem 2 (`ReluSplitter`(Conv) Gadget is Exact)**.** *For any convolutional filter $\mathbf{f}_o$ with kernel $\mathbf{k}_o$, bias $\mathbf{b}_o$, and post-ReLU output channel $\mathbf{f}_o(X)$, the `ReluSplitter` split gadget (comprising filters $\mathbf{f}_a$, $\mathbf{f}_b$, and $\mathbf{f}_m$ as defined in §IV, using any split parameters $s_{pos} > 0$, $s_{neg} < 0$, and offset τ) produces an identical output channel $\mathbf{f}_m(X)$ for any input tensor X:*

$$\mathbf{f}_m(X) = \mathbf{f}_o(X) \quad \textit{for all } X.$$

Proof (Sketch). The proof proceeds by demonstrating that for any arbitrary spatial location (i, j) in the output channel, the output of the `ReluSplitter`(Conv) gadget is identical to that of the original filter. This local equivalence is established by reusing the proof from Theorem 1, and the overall proof is completed by generalizing this finding to the entire output channel.

As shown in §IV-A, the forward-facing filters, $\mathbf{f}_a$ and $\mathbf{f}_b$, inherit their kernel size, stride, and padding from the original filter $\mathbf{f}_o$, differing only in their scaled weights and adjusted biases. This structural inheritance ensures that the output channel produced by $\mathbf{f}_a$ and $\mathbf{f}_b$ have the same dimensions and spatial correspondence as that of $\mathbf{f}_o$. Moreover, since $\mathbf{f}_m$ only performs 1×1 convolution, the dimension and spatial correspondence is further preserved in $\mathbf{f}_m(X)$.

Let $\mathbf{f}_o(X)_{i,j}$ refer to the single scalar value at the spatial location (i, j) within the output channel $\mathbf{f}_o(X)$, and let $X_{\mathbf{f}_o[i,j]}$ denote the input patch in X from which $\mathbf{f}_o(X)_{i,j}$ is computed. The spatial correspondence implies that for arbitrary location (i, j), the outputs $\mathbf{f}_o(X)_{i,j}$, $\mathbf{f}_a(X)_{i,j}$, and $\mathbf{f}_b(X)_{i,j}$ are all computed using the same patch. That is:

$$X_{\mathbf{f}_o[i,j]} = X_{\mathbf{f}_a[i,j]} = X_{\mathbf{f}_b[i,j]}$$

With this notation, we can rewrite the pre-activation value of $\mathbf{f}_o(X)$ at each spatial location, $z_o(X)_{i,j}$, as:

$$z_o(X)_{i,j} = \mathbf{k}_o \cdot X_{\mathbf{f}_o[i,j]} + \mathbf{b}_o$$

This operation is identical to the FC neuron's pre-activation, where the kernel $\mathbf{k}_o$ and the patch $X_{i,j}$ act as the weight and input vectors. Similarly, we have:

$$z_m(X)_{i,j} = \frac{1}{s_{pos}} \mathbf{f}_a(X)_{i,j} + \frac{1}{s_{neg}} \mathbf{f}_b(X)_{i,j} + \tau$$

where

$$\mathbf{f}_a(X)_{i,j} = ReLU(s_{pos} \mathbf{k}_o \cdot X_{\mathbf{f}_a[i,j]} + s_{pos}(\mathbf{b}_o - \tau))$$
$$\mathbf{f}_b(X)_{i,j} = ReLU(s_{neg} \mathbf{k}_o \cdot X_{\mathbf{f}_b[i,j]} + s_{neg}(\mathbf{b}_o - \tau))$$

After applying the necessary algebraic simplification, we can reuse the case analysis from Theorem 1 to show $z_o(X)_{i,j} = z_m(X)_{i,j}$ at arbitrary location (i, j). Since this equivalence holds across the entire output channel, it follows that $z_o(X) = z_m(X)$. Since both filters apply the same ReLU activation, their post-activation outputs are also identical: $\mathbf{f}_m(X) = \mathbf{f}_o(X)$, which proves Theorem 2. □

Theorem 3 (`ReluSplitter` Forward Gadget is Exact)**.** *For any fully connected neuron n_o or convolutional filter $\mathbf{f}_o$, their respective `ReluSplitter` forward gadgets, n_f and $\mathbf{f}_f$, produce outputs identical to the original components for all inputs:*

$$n_f(x) = n_o(x) \quad \textit{for all } x,$$

$$\mathbf{f}_f(X) = \mathbf{f}_o(X) \quad \textit{for all } X.$$

Proof (Sketch). The theorem holds by the construction of the forwarding gadget, as defined in §IV-B. We examine the fully connected and convolutional cases separately.

Fully Connected: The forward gadget n_f is a neuron whose input is the output vector from the layer containing n_o. By construction, its weights are all 0 except for a single weight

Tab. I: Benchmark instances. α: $\alpha\beta$-CROWN, nsat: NeuralSAT, n: nnenum, m: Marabou.

Benchmark	Type	Neuron	Instance	Timeout	Verifier
ACAS Xu	FC	300	156	60s	α,nsat,n,m
MNIST_FC	FC	0.5-1.5K	41	120s	α,nsat,n,m
Oval21	CNN	0.6-2K	86	120s	α,nsat
SRI ResNet A/B	ResNet+CNN	11K	67	120s	α,nsat

of 1 corresponding to the input from n_o, and its bias is 0. Therefore, its output is exactly $1 \cdot n_o(x) + 0 = n_o(x)$.

Convolutional: The forward gadget $\mathbf{f}_f$ is a 1×1 convolution. Its kernel has a weight of 1 for the input channel produced by the original filter $\mathbf{f}_o$ and 0 for all other channels, with a bias of 0. Since a 1×1 convolution operates on each spatial location independently, the output at any location is simply $1 \cdot \mathbf{f}_o(X)_{i,j}$. The resulting tensor is therefore identical to $\mathbf{f}_o(X)$.

In both configurations, the gadget is constructed to be an identity operation for the specific component it forwards, thus proving the theorem. □

V. Evaluation

ReluSplitter is implemented in about 3000 LOCs in Python. It supports the standard ONNX [39] format for representing neural networks and VNN-LIB [40] for properties. The tool has few dependencies and uses standard libraries including PyTorch [41], onnx2pytorch [42] for model manipulation and conversion, and the well-known auto-LiRPA [38] library for computing abstractions.

Our evaluation focuses on demonstrating the effectiveness of ReluSplitter in generating challenging verification instances. We aim to show that ReluSplitter can transform DNN verification instances that are easily solved by existing DNN verifiers into more difficult ones.

a) Benchmarks and Verifiers: We used four ReLU-based benchmarks from VNN-COMP [17], covering a variety of NN architectures including fully connected (FC) NNs, convolutional NNs (CNN), and CNNs with residual connections (ResNet). We restricted our selection to benchmarks from VNN-COMP to ensure compatibility and strong support across the evaluated verifiers. The selected benchmarks are ACAS Xu, MNIST_FC, Oval 21, and SRI_ResNet, each summarized in Tab. I. ACAS Xu includes 10 safety properties across 45 ReLU-based FC networks, each with 300 neurons over 6 layers, designed to issue aircraft collision avoidance advisories. MNIST_FC comprises three FC ReLU networks for digit classification, each taking 28x28 grayscale images as input and producing 10 outputs; they share a hidden layer width of 256 but differ in depth (2, 4, and 6 layers). Oval 21 and SRI_ResNet focus on CNNs trained on CIFAR-10 [43], with the latter adopting a ResNet architecture.

We used the latest versions of four SOTA DNN verifiers: $\alpha\beta$-CROWN [29], [33], Marabou [32], NeuralSAT [8], [9], [31], and nnenum [30], [44]. Since Marabou and nnenum often struggle with CNNs (frequently timing out) we exclude them from the evaluation on Oval 21 and SRI_ResNet.

Experiments were conducted on a Linux machine with an AMD Threadripper 64-core 4.2GHz CPU, 128GB RAM, and an NVIDIA GeForce RTX 4090 GPU 24GB VRAM. The full set of experiments took approximately seven days to complete. The generation time for each instance was negligible (less than 5 seconds) compared to the verification time.

A. Experiment Design

We first ran an initial verification on all benchmark instances using four verifiers to filter out hard cases and retained only those that could be solved within a reduced timeout (Tab. I). These easier cases became our **original instances**. From each original, we created two modified versions: a **`ReluSplitter` instance**, designed to increase difficulty, and a **baseline instance**, designed to match size but preserve stability. The modified versions used a timeout that was three times longer (e.g., ACAS Xu uses a 60s timeout for original instances, and 180s for both the baseline and ReluSplitter instances).

For the ReluSplitter instances, we split stably active neurons in the first ReLU layer of FC networks, and all kernels in the first ReLU layer of CNNs. This keeps the changes impactful but manageable. To create the baseline instance, we apply the same structural transformation—replacing the same set of neurons or kernels with split gadgets—but with modified parameters that avoid destabilization. Specifically, s_{neg} and s_{pos} are randomly chosen from the range $(0, 1)$, subject to the constraint $s_{neg} + s_{pos} = 1$. The split offset τ is fixed to 0 and the backward-facing component uses a fixed weight of 1 for each input from the forward-facing component. This ensures that the baseline matches the size and structure of the ReluSplitter instance without introducing additional unstable neurons.

We then ran all three versions (original, ReluSplitter, and baseline) through the verifiers, repeating the process five times with different random seeds and using the median verification time to compare slowdowns.

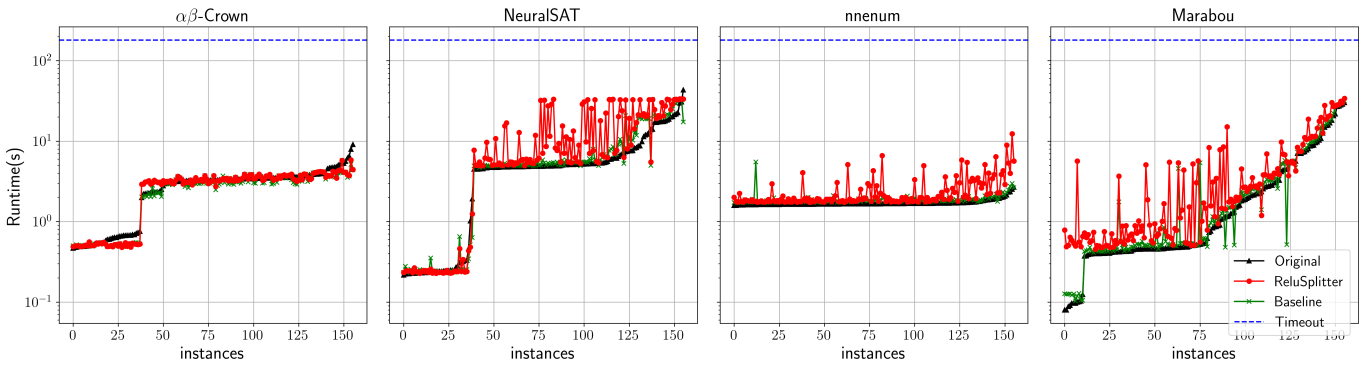
B. Results

Tab. II reports the slowdown induced by the baseline instances, ReluSplitter instances, and the net slowdown (Δ_{SD}), which reports the difference in slowdown between the matching baseline and tool instances. It reflects how much additional slowdown ReluSplitter introduces beyond what is caused by the increase in network size alone. The slowdown and net slowdown are computed as:

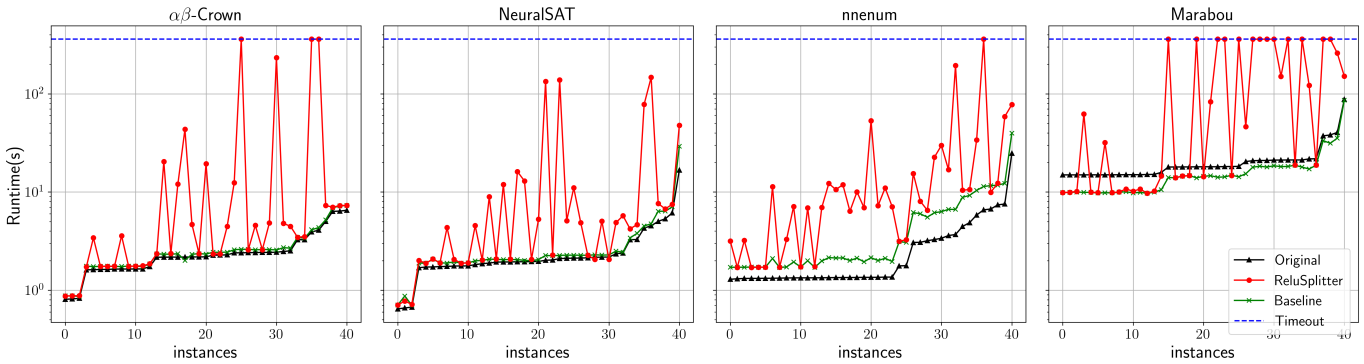
$$\text{Slowdown} = \frac{\text{Verification Time (generated instance)}}{\text{Verification Time (original instance)}} - 1$$

$$\Delta_{SD} = \text{Slowdown}_{\texttt{ReluSplitter}} - \text{Slowdown}_{\text{baseline}}$$

The table reports the mean, median, and maximum net slowdown observed for each benchmark–verifier pair. The **Instance** column indicates the number of original instances in



(a) ACAS Xu



(b) MNIST_FC

Fig. 5: Runtime of four verifiers on FC benchmarks. The y-axis (Runtime) is logarithmic; the x-axis represents instances, sorted by the verification time on the original instance.

the benchmark. For example, four verifiers were tested on the `MNIST_FC` benchmark, which includes 41 original instances. Using $\alpha\beta$-`CROWN` on this benchmark, `ReluSplitter` induced a 11.61x average, 0.10x median, and 148.78x maximum slowdown.

Fig. 5 and Fig. 6 show the verification time of the original, `ReluSplitter`, and baseline instances across different benchmarks and verifiers. The x-axis represents individual instances, sorted by the verification time of their corresponding original instance. The y-axis shows verification time on a logarithmic scale. Each marker corresponds to one instance: black triangle for the original, green cross for the baseline, and red dot for the `ReluSplitter` instance. The horizontal blue dashed line indicates the timeout threshold.

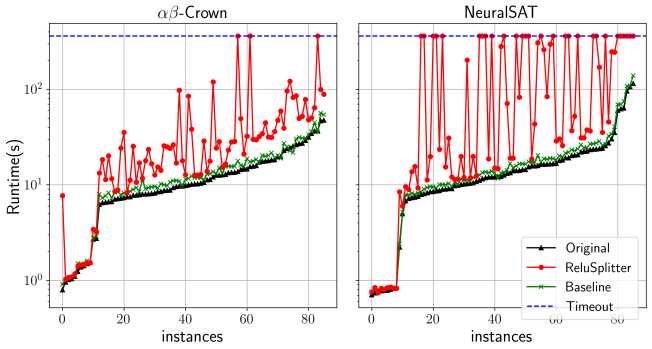
Additionally, no conflicting verification results were observed (i.e. the verifier reported SAT on the original instance but reported UNSAT on `ReluSplitter` or baseline instance, or vice-versa).

a) `ReluSplitter` *effectively increases verification difficulty:* Our results show that instances modified by `ReluSplitter` often require significantly more time to verify. While some experience substantial slowdowns, others see only moderate increases in runtime. Notably, harder instances tend to incur greater slowdowns, aligning with our expectation since verifier's search tree grows exponentially with the number of unstable neurons in the network. These unstable neurons force verifiers to perform more branching and abstraction, increasing the computational burden. In contrast, easy instances—particularly those that terminate quickly—exhibit slowdowns that are far less pronounced. This is because their runtimes are largely dominated by fixed overheads associated with the verifier, such as library loading, hardware initialization, memory allocation, and ONNX/VNNLIB parsing, rather than the complexity of the network itself.

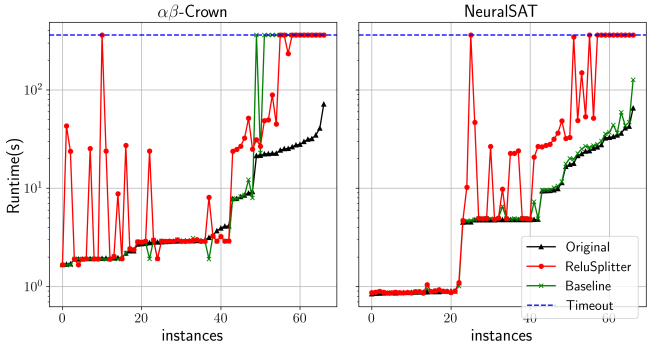
This trend is also reflected in the distribution of slowdown values across our experiments. Although the mean and maximum slowdowns introduced by `ReluSplitter` are considerably high across most verifier-benchmark pairs, the median slowdown remains much lower. This discrepancy is likely due to our experimental design, where we intentionally selected a set of "easy" instances but did not exclude trivially solvable ones. As a result, `ReluSplitter` has minimal impact on many of these simple cases, which is evident from the concentration of data points near the origin on the x-axis in Fig. 5 and Fig. 6. However, as instance difficulty increases, the impact of `ReluSplitter` becomes more substantial, often resulting in dramatic increases in verification time — a trend that is clearly visible in the rightward regions of both

Tab. II: Slowdown (avg./med./max.) for *baseline*, `ReluSplitter`, and *net slowdown* (Δ_{SD}), per benchmark and verifier.

Benchmark	Instance	Verifier	Slowdown (avg./med./max.) Baseline	`ReluSplitter`	Δ_{SD}
ACAS Xu	156	$\alpha\beta$-CROWN	-0.07x / -0.08x / 0.22x	-0.02x / -0.00x / 0.46x	0.06x / 0.02x / 0.51x
		Marabou	0.25x / 0.11x / 9.28x	1.84x / 0.42x / 55.59x	1.59x / 0.32x / 55.40x
		NeuralSAT	0.13x / 0.07x / 1.80x	0.85x / 0.19x / 5.72x	0.72x / 0.09x / 5.62x
		nnenum	0.10x / 0.07x / 2.40x	0.41x / 0.15x / 3.91x	0.31x / 0.04x / 3.74x
MNIST_FC	41	$\alpha\beta$-CROWN	0.07x / 0.07x / 0.13x	11.61x / 0.10x / 148.78x	11.54x / 0.02x / 148.69x
		Marabou	-0.23x / -0.22x / -0.02x	5.59x / 0.72x / 19.04x	5.82x / 0.73x / 19.26x
		NeuralSAT	0.11x / 0.07x / 0.74x	5.39x / 0.29x / 65.16x	5.28x / 0.25x / 65.04x
		nnenum	0.58x / 0.58x / 1.02x	6.49x / 3.73x / 53.66x	5.91x / 3.01x / 52.93x
Oval21	86	$\alpha\beta$-CROWN	0.12x / 0.13x / 0.42x	1.99x / 0.87x / 25.16x	1.86x / 0.79x / 24.87x
		NeuralSAT	0.11x / 0.12x / 0.28x	8.83x / 1.63x / 44.52x	8.72x / 1.49x / 44.45x
SRI ResNet A/B	67	$\alpha\beta$-CROWN	3.05x / 0.01x / 15.83x	6.23x / 0.06x / 185.23x	3.17x / 0.00x / 185.25x
		NeuralSAT	0.07x / 0.03x / 0.94x	3.94x / 0.88x / 78.40x	3.87x / 0.68x / 78.37x



(a) Oval21



(b) SRI ResNet A/B

Fig. 6: Runtime of two verifiers on CNN benchmarks. The y-axis (Runtime) is logarithmic; the x-axis represents instances, sorted by the verification time on the original instance.

figures. These results support our hypothesis that destabilizing neurons can significantly increase problem complexity, especially when applied to non-trivial networks. To further validate this, we re-ran the 55 `ReluSplitter` instances from `Oval 21` and `SRI_ResNet` that had timed out for $\alpha\beta$-CROWN or `NeuralSAT` using a 10x longer (20-minute) timeout; only 7 of them became solvable, confirming the substantial complexity increase from `ReluSplitter`.

b) `ReluSplitter` can provide insight into verifier evaluation: Our experimental results demonstrate that each verifier exhibits distinct behavior when solving `ReluSplitter` instances. On the `ACAS Xu` benchmark, all verifiers were able to scale, though some experienced more significant slowdowns than others. For example, `Marabou` had a mean slowdown of 1.84x and an median of 0.42x, which is more than 2 times higher than any other verifiers.

These differences become more pronounced on more complex benchmarks such as `MNIST_FC`, which involves deeper and larger networks. On this benchmark, `NeuralSAT` demonstrated the most robust performance, successfully verifying all 41 `ReluSplitter` instances . `nnenum` performed similarly well, timing out on only 1 instance. In contrast, $\alpha\beta$-CROWN encountered 3 timeouts, while `Marabou` struggled significantly, with 13 timeouts on `ReluSplitter` instances.

These differences are difficult to observe with the original instances, as all verifiers are able to solve them. By increasing instance difficulty while preserving ground truth, `ReluSplitter` highlights differences in verifier scalability that are otherwise obscured. This uneven distribution of instance difficulty has been a long-standing challenge in DNN verification. Many benchmarks are either too easy—solvable by all verifiers—or too hard—unsolvable by any verifier [17].

In some extreme cases, the slowdown of `ReluSplitter` instances can exceed 100x (e.g., $\alpha\beta$-CROWN on `SRI_ResNet`, with a max slowdown of 185x). These typically come from easy instances where `ReluSplitter` is able to introduce substantial slowdown; For harder instances, the slowdown is often capped by the timeout threshold, so the true slowdown may be underestimated.

`ReluSplitter` also reveals insights into the effectiveness of different decision heuristics. For example, both $\alpha\beta$-CROWN and `NeuralSAT` rely on `auto-LiRPA` for bound computation but differ in their branching strategies. `NeuralSAT` adapts the DPLL(T) algorithm [8], [31], while $\alpha\beta$-CROWN implements several heuristics [29], [33], [45]. Although their overall trends are similar across benchmarks, notable differ-

ences emerge. On Oval 21, NeuralSAT timed out on many ReluSplitter instances that $\alpha\beta$-CROWN successfully verified. On SRI_ResNet, $\alpha\beta$-CROWN experienced significant slowdowns on easier instances, while performing better on medium-difficulty ones. NeuralSAT showed the opposite pattern—minimum slowdown on easier instances but larger slowdown on medium and hard ones.

These results suggest that ReluSplitter not only challenges verifiers in a controlled way, but also helps researchers better understand verifier behavior, strengths, and weaknesses.

c) ReluSplitter can help expose potential flaws in verifiers: In addition to increasing instance difficulty, ReluSplitter helps uncover subtle issues in verifier implementations. For example, nnenum occasionally crashes on ReluSplitter instances, despite successfully verifying the original and baseline instances. Since all other verifiers handled these instances without error, this points to potential bugs or unhandled edge cases in nnenum.

ReluSplitter also reveals unexpected performance patterns that may indicate inefficiencies or opportunities for optimization. On the MNIST_FC benchmark, Marabou consistently verifies baseline instances faster than the original ones—opposite the trend observed with other verifiers. This could suggest that certain layer patterns introduced in the baseline transformation inadvertently align better with Marabou 's heuristics or constraint encodings. Additionally, on SRI_ResNet, we find that $\alpha\beta$-CROWN times out on some baseline instances, even though it solves both the original and ReluSplitter versions efficiently. These counterintuitive behaviors suggest that ReluSplitter can surface behaviors worth further investigation.

Thus ReluSplitter is not just a stress test tool for verifiers. It also helps identify potential flaws and optimization opportunities that standard benchmarks may overlook.

Our evaluation shows that ReluSplitter substantially increases verification difficulty—by up to 185.23x. This makes differences in verifier performance and scalability more apparent than with standard benchmarks, and enabling a clearer, more comprehensive assessment of tool behavior across a wide range of instance complexities.

VI. Related Work

Verification tools such as $\alpha\beta$-CROWN [29], [33], NeuralSAT [8], [9], [31], Marabou [11], [32], nnenum [44], and NNV [46] represent a range of modern approaches to neural network verification. Despite their differences, these tools largely follow the Branch-and-Bound (BaB) framework, which recursively splits neurons and uses abstract domains to over-approximate bounds. Commonly used domains include interval arithmetic (IBP) [37], [47], zonotopes [48], polyhedra [49], star sets [50], and CROWN-based bounds [29]. The performance of these verifiers depends heavily on how abstract domains are combined with tightening methods and BaB strategies, and ongoing research continues to investigate which combinations are most effective for scalable, sound verification.

Benchmark generation is a well-developed direction in many fields, such as software engineering [51]–[53], SAT/SMT solving [54]–[57], and, more recently, AI [58]–[60]. In neural network verification (DNNV), *R4V* [21] and *GDVB* [20] are among the earliest attempts at benchmark generation. *R4V* uses knowledge distillation to downscale a seed DNNV instance in order to boost verifiability. *GDVB* builds on *R4V* to create more diverse DNNV benchmarks by altering the seed DNNV instance. However, both rely on training data and knowledge distillation, often producing instances with unknown ground truth. This makes it difficult to assess verifier correctness

Another recent work [19] explored the generation of soundness benchmarks for DNN verification. The authors identified two major limitations in existing benchmarks. First, many benchmarks lack known ground truth, making it difficult to assess verifier soundness. Second, they observed that many existing DNNV benchmarks are unintentionally too easy, as their properties can often be falsified using standard adversarial attack methods rather than requiring full verification. The authors then proposed a specialized training procedure that embeds hidden counterexamples directly into the network. This ensures the resulting benchmark instances have known *SAT* solutions while posing significant challenges to verifiers.

VII. Conclusion

To address the challenge of benchmark generation for DNN verification, we introduced ReluSplitter, a technique for generating harder DNNV benchmarks from easy ones by destabilizing ReLU neurons. ReluSplitter fills a critical gap in the evaluation pipeline by producing a spectrum of problem instances that are more difficult yet still semantically equivalent, enabling more robust benchmarking and providing deeper insight into verifier strengths and limitations. Our experiments across four standard benchmarks and four state-of-the-art verifiers demonstrate that ReluSplitter consistently increases problem difficulty, reveals meaningful differences in verifier performance, and uncovers potential implementation bugs and heuristic behaviors.

Future directions include extending ReluSplitter to support other ReLU variants such as Leaky ReLU, handling a broader range of architectures, and synthesizing benchmarks from networks with smooth activations like sigmoid and tanh. Furthermore, improving the heuristic strategies for choosing split candidates and parameters may yield stronger and more efficient instance generation.

VIII. Acknowledgment

We thank the anonymous reviewers for their helpful comments. This material is based in part upon work supported by the National Science Foundation under grant numbers 2422036, 2319131, 2238133, and 2200621, and by an Amazon Research Award.

REFERENCES

[1] F. Yu, Z. Qin, C. Liu, D. Wang, and X. Chen, "Rein the robuts: Robust dnn-based image recognition in autonomous driving systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 6, pp. 1258–1271, 2020.

[2] M. Cococcioni, F. Rossi, E. Ruffaldi, S. Saponara, and B. D. de Dinechin, "Novel arithmetics in deep neural networks signal processing for autonomous driving: Challenges and opportunities," *IEEE Signal Processing Magazine*, vol. 38, no. 1, pp. 97–110, 2020.

[3] Y. Huang and Y. Chen, "Autonomous driving with deep learning: A survey of state-of-art technologies," *arXiv preprint arXiv:2006.06091*, 2020.

[4] J. Shi, X. Fan, J. Wu, J. Chen, and W. Chen, "Deepdiagnosis: Dnn-based diagnosis prediction from pediatric big healthcare data," in *2018 Sixth International Conference on Advanced Cloud and Big Data (CBD)*. IEEE, 2018, pp. 287–292.

[5] V. Sharma, A. Rasool, and G. Hajela, "Prediction of heart disease using dnn," in *2020 second international conference on inventive research in computing applications (ICIRCA)*. IEEE, 2020, pp. 554–562.

[6] S. Hamida, O. El Gannour, Y. Lamalem, S. Saleh, D. Lamrani, and B. Cherradi, "Efficient medical diagnosis hybrid system based on rf-dnn mixed model for skin diseases classification," in *2023 3rd International Conference on Innovative Research in Applied Science, Engineering and Technology (IRASET)*. IEEE, 2023, pp. 01–08.

[7] C. Brix, S. Bak, T. T. Johnson, and H. Wu, "The Fifth International Verification of Neural Networks Competition (VNN-COMP 2024): Summary and Results," *https://www.arxiv.org/pdf/2412.19985*, 2024.

[8] H. Duong, T. Nguyen, and M. B. Dwyer, "Neuralsat: A high-performance verification tool for deep neural networks," in *International Conference on Computer Aided Verification*. Springer, 2025, pp. 409–423.

[9] H. Duong, D. Xu, T. Nguyen, and M. B. Dwyer, "Harnessing neuron stability to improve dnn verification," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 859–881, 2024.

[10] D. Zhou, C. Brix, G. A. Hanasusanto, and H. Zhang, "Scalable neural network verification with branch-and-bound inferred cutting planes," in *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[11] H. Wu, O. Isac, A. Zeljić, T. Tagomori, M. Daggitt, W. Kokke, I. Refaeli, G. Amir, K. Julian, S. Bassan *et al.*, "Marabou 2.0: a versatile formal analyzer of neural networks," in *International Conference on Computer Aided Verification*. Springer, 2024, pp. 249–264.

[12] M. N. Müller, G. Makarchuk, G. Singh, M. Püschel, and M. Vechev, "Prima: general and precise neural network certification via scalable convex hull approximations," *Proceedings of the ACM on Programming Languages*, vol. 6, no. POPL, pp. 1–33, 2022.

[13] S. Bak, T. Dohmen, K. Subramani, A. Trivedi, A. Velasquez, and P. Wojciechowski, "The hexatope and octatope abstract domains for neural network verification," *Formal Methods in System Design*, pp. 1–22, 2024.

[14] S. Demarchi, D. Guidotti, L. Pulina, and A. Tacchella, "Never2: learning and verification of neural networks," *Soft Computing*, vol. 28, no. 19, pp. 11 647–11 665, 2024.

[15] C. Brix, M. N. Müller, S. Bak, T. T. Johnson, and C. Liu, "First three years of the international verification of neural networks competition (vnn-comp)," *International Journal on Software Tools for Technology Transfer*, vol. 25, no. 3, pp. 329–339, 2023.

[16] S. Bak, C. Liu, and T. Johnson, "The second international verification of neural networks competition (vnn-comp 2021): Summary and results," *arXiv preprint arXiv:2109.00498*, 2021.

[17] M. N. Müller, C. Brix, S. Bak, C. Liu, and T. T. Johnson, "The third international verification of neural networks competition (vnn-comp 2022): Summary and results," *arXiv preprint arXiv:2212.10376*, 2022.

[18] C. Brix, S. Bak, C. Liu, and T. T. Johnson, "The fourth international verification of neural networks competition (vnn-comp 2023): Summary and results," *arXiv preprint arXiv:2312.16760*, 2023.

[19] X. Zhou, H. Xu, A. Xu, Z. Shi, C.-J. Hsieh, and H. Zhang, "Testing neural network verifiers: A soundness benchmark with hidden counterexamples," *arXiv preprint arXiv:2412.03154*, 2024.

[20] D. Xu, D. Shriver, M. B. Dwyer, and S. Elbaum, "Systematic generation of diverse benchmarks for dnn verification," in *International Conference on Computer Aided Verification*. Springer, 2020, pp. 97–121.

[21] D. Shriver, D. Xu, S. Elbaum, and M. B. Dwyer, "Refactoring neural networks for verification," *arXiv preprint arXiv:1908.08026*, 2019.

[22] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An efficient smt solver for verifying deep neural networks," in *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I 30*. Springer, 2017, pp. 97–117.

[23] R. R. Bunel, I. Turkaslan, P. Torr, P. Kohli, and P. K. Mudigonda, "A unified view of piecewise linear neural network verification," *Advances in neural information processing systems*, vol. 31, 2018.

[24] C. Liu, T. Arnon, C. Lazarus, and M. J. Kochenderfer, "Neuralverification. jl: algorithms for verifying deep neural networks," in *ICLR 2019 Debugging Machine Learning Models Workshop*, 2019.

[25] Y. Zhong, R. Wang, and S.-C. Khoo, "Expediting neural network verification via network reduction," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 1263–1275.

[26] K. Y. Xiao, V. Tjeng, N. M. Shafiullah, and A. Madry, "Training for faster adversarial robustness verification via inducing relu stability," *arXiv preprint arXiv:1809.03008*, 2018.

[27] D. Xu, N. J. Mozumder, H. Duong, and M. B. Dwyer, "Training for verification: Increasing neuron stability to scale dnn verification," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2024, pp. 24–44.

[28] Z. Liu, Z. Zhao, F. Song, J. Sun, P. Yang, X. Huang, and L. Zhang, "Training verification-friendly neural networks via neuron behavior consistency," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 39, no. 6, 2025, pp. 5757–5765.

[29] S. Wang, H. Zhang, K. Xu, X. Lin, S. Jana, C.-J. Hsieh, and J. Z. Kolter, "Beta-CROWN: Efficient bound propagation with per-neuron split constraints for complete and incomplete neural network verification," *Advances in Neural Information Processing Systems*, vol. 34, 2021.

[30] S. Bak, H.-D. Tran, K. Hobbs, and T. T. Johnson, "Improved geometric path enumeration for verifying relu neural networks," in *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part I 32*. Springer, 2020, pp. 66–96.

[31] H. Duong, T. Nguyen, and M. Dwyer, "A dpll(t) framework for verifying deep neural networks," 2024.

[32] G. Katz, D. A. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljić *et al.*, "The marabou framework for verification and analysis of deep neural networks," in *International Conference on Computer Aided Verification*. Springer, 2019, pp. 443–452.

[33] Z. Shi, Q. Jin, Z. Kolter, S. Jana, C.-J. Hsieh, and H. Zhang, "Neural network verification with branch-and-bound for general nonlinearities," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2025.

[34] C. Ferrari, M. N. Muller, N. Jovanovic, and M. Vechev, "Complete verification via multi-neuron relaxation guided branch-and-bound," *arXiv preprint arXiv:2205.00263*, 2022.

[35] R. Bunel, P. Mudigonda, I. Turkaslan, P. Torr, J. Lu, and P. Kohli, "Branch and bound for piecewise linear neural network verification," *Journal of Machine Learning Research*, vol. 21, no. 2020, 2020. [Online]. Available: https://dl.acm.org/doi/10.5555/3455716.3455758

[36] T. Chen, H. Zhang, Z. Zhang, S. Chang, S. Liu, P.-Y. Chen, and Z. Wang, "Linearity grafting: Relaxed neuron pruning helps certifiable robustness," in *International Conference on Machine Learning*. PMLR, 2022, pp. 3760–3772.

[37] S. Gowal, K. Dvijotham, R. Stanforth, R. Bunel, C. Qin, J. Uesato, R. Arandjelovic, T. Mann, and P. Kohli, "On the effectiveness of interval bound propagation for training verifiably robust models," *arXiv preprint arXiv:1810.12715*, 2018.

[38] K. Xu, Z. Shi, H. Zhang, Y. Wang, K.-W. Chang, M. Huang, B. Kailkhura, X. Lin, and C.-J. Hsieh, "Automatic perturbation analysis for scalable certified robustness and beyond," *Advances in Neural Information Processing Systems*, vol. 33, 2020.

[39] "Open neural network exchange (onnx)," 2019, accessed: 2024-10-05. [Online]. Available: https://onnx.ai/

[40] "Vnn-lib: Verification of neural networks benchmarks library," 2020, accessed: 2024-10-05. [Online]. Available: https://www.vnnlib.org/

[41] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An

imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.
[42] E. developers, I. Kalgin, A. Yanchenko, P. Ivanov, and A. Goncharenko, "onnx2torch," https://enot.ai/, 2021, version: x.y.z.
[43] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.
[44] S. Bak, "nnenum: Verification of relu neural networks with optimized abstraction refinement," in *NASA Formal Methods Symposium*. Springer, 2021, pp. 19–36.
[45] H. Zhang, S. Wang, K. Xu, Y. Wang, S. Jana, C.-J. Hsieh, and Z. Kolter, "A branch and bound framework for stronger adversarial attacks of ReLU networks," in *Proceedings of the 39th International Conference on Machine Learning*, vol. 162, 2022, pp. 26 591–26 604.
[46] H.-D. Tran, X. Yang, D. Manzanas Lopez, P. Musau, L. V. Nguyen, W. Xiang, S. Bak, and T. T. Johnson, "Nnv: the neural network verification tool for deep neural networks and learning-enabled cyber-physical systems," in *International conference on computer aided verification*. Springer, 2020, pp. 3–17.
[47] M. Mirman, T. Gehr, and M. Vechev, "Differentiable abstract interpretation for provably robust neural networks," in *International Conference on Machine Learning*. PMLR, 2018, pp. 3578–3586.
[48] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev, "Ai2: Safety and robustness certification of neural networks with abstract interpretation," in *2018 IEEE symposium on security and privacy (SP)*. IEEE, 2018, pp. 3–18.
[49] G. Singh, T. Gehr, M. Püschel, and M. Vechev, "An abstract domain for certifying neural networks," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–30, 2019.
[50] H.-D. Tran, D. Manzanas Lopez, P. Musau, X. Yang, L. V. Nguyen, W. Xiang, and T. T. Johnson, "Star-based reachability analysis of deep neural networks," in *Formal Methods–The Next 30 Years: Third World Congress, FM 2019, Porto, Portugal, October 7–11, 2019, Proceedings 3*. Springer, 2019, pp. 670–686.
[51] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.
[52] G. Amendola, F. Ricca, M. Truszczynski *et al.*, "A generator of hard 2qbf formulas and asp programs." in *KR*, 2018, pp. 52–56.
[53] C. Zhang and Z. Su, "Smt2test: From smt formulas to effective test cases," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA2, pp. 222–245, 2024.
[54] T. Balyo and L. Chrpa, "Using algorithm configuration tools to generate hard sat benchmarks," in *Proceedings of the International Symposium on Combinatorial Search*, vol. 9, no. 1, 2018, pp. 133–137.
[55] D. Winterer, C. Zhang, and Z. Su, "Validating smt solvers via semantic fusion," in *Proceedings of the 41st ACM SIGPLAN Conference on programming language design and implementation*, 2020, pp. 718–730.
[56] J. Park, D. Winterer, C. Zhang, and Z. Su, "Generative type-aware mutation for testing smt solvers," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–19, 2021.
[57] M. Lauria, J. Elffers, J. Nordström, and M. Vinyals, "Cnfgen: A generator of crafted benchmarks," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2017, pp. 464–473.
[58] L. Doan and T. Nguyen, "Ai-assisted autoformalization of combinatorics problems in proof assistants," in *2025 IEEE/ACM 47th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 2025, pp. 1–5.
[59] C. Loughridge, Q. Sun, S. Ahrenbach, F. Cassano, C. Sun, Y. Sheng, A. Mudide, M. R. H. Misu, N. Amin, and M. Tegmark, "Dafnybench: A benchmark for formal software verification," *arXiv preprint arXiv:2406.08467*, 2024.
[60] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, "Fuzzbench: an open fuzzer benchmarking platform and service," in *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, 2021, pp. 1393–1403.