ARENA: Enhancing Abstract Refinement for Neural Network Verification

Yuyi Zhong   Quang-Trung Ta   Siau-Cheng Khoo

{yuyizhong, taqt, khoosc}@comp.nus.edu.sg

School of Computing, National University of Singapore, Singapore

Abstract. As neural networks have taken on a critical role in real-world applications, formal verification is earnestly needed to guarantee the safety properties of the networks. However, it remains challenging to balance the trade-off between precision and efficiency in abstract interpretation based verification methods. In this paper, we propose an abstract refinement process that leverages the convex hull techniques to improve the analysis efficiency. Specifically, we introduce the double description method in the convex polytope domain to detect and eliminate multiple *spurious* adversarial labels simultaneously. We also combine the new activation relaxation technique with the iterative abstract refinement method to compensate for the precision loss during abstract interpretation. We have implemented our proposal into a verification framework named ARENA, and assessed its effectiveness by conducting a series of experiments. These experiments show that ARENA yields significantly better verification precision compared to the existing abstract-refinement-based tool DeepSRGR. It also identifies falsification by detecting adversarial examples, with reasonable execution efficiency. Lastly, it verifies more images than the state-of-the-art verifiers PRIMA (CPU-based) and α, β-CROWN (GPU-based).

Keywords: Abstract Refinement · Double Description Method · Neural Network Verification.

1 Introduction

As neural networks have been proverbially applied to safety-critical systems, formal guarantee about the safety properties of the networks, such as robustness, fairness, etc., is earnestly needed. For example, researchers have been working on robustness verification of neural networks, to ascertain that the network classification result can remain the same when the input image is perturbed subtly and imperceptibly during adversarial attacks [1,2].

There exists sound and complete verification techniques where the robustness property can be ascertained but regrettably at high complexity and execution cost [3,4]. For better scalability, several incomplete verifiers have been proposed to analyze larger networks with abstract interpretation technique while bearing exactness sacrifices [5–7]. To mitigate this shortcoming, there have been investigations into better convex relaxation [8,9] or iterative abstract refinement [10] to make up with the precision loss in abstract interpretation techniques.

This work is inspired by the counterexample guided abstraction refinement (CEGAR) method [11] in program analysis, aiming to improve the precision of abstract interpretation results by identifying *spurious counterexamples*: these are examples which appear to have violated desired analysis outcome – due to over-approximated calculation inherent in the abstract interpretation computation – but can be shown to be fake by the refinement method. Proof of the existence of spurious counterexamples can diminish the range of inconclusive results produced by abstract interpretation. In the context of neural network verification, such spurious counterexamples can be conceptualized as adversarial *regions* that are perceived to have lent support (spuriously) on certain adversarial labels; ie., labels which differ from the designated label in the robustness test.

An existing work that has successfully employed abstract refinement technique to improve the precision of the abstract-interpretation based verification tool is DeepSRGR [10]. That work repetitively selects an adversarial label and attempts to eliminate the corresponding spurious region progressively through iteration of refinements. Technically, it encodes a spurious region as a linear inequality, adds it to the constraint encoding of the network, and employs linear programming with the objective set to optimize the concrete bounds of selected ReLU neurons in the network. This process is repeated until either the spurious region is found to be inconsistent with the encoded network, or time out.

In this paper, we enhance the existing effectiveness of DeepSRGR by introducing *convex hull* techniques (ie., techniques that observe and conform to convex property) to abstract refinement computation. Together, these techniques facilitate *simultaneous* elimination of multiple spurious regions, and capitalize the *dependencies* among ReLU neurons. Specifically, we tighten the looseness of ReLU approximation during abstract refinement process through a *mutli-ReLU convex abstraction technique* (*cf.* [9]) that captures dependencies within a set of ReLU neurons. Moreover, we leverage a double-description method (*cf.* [12]) used in convex polytope computation to eliminate multiple spurious regions simultaneously; this circumvents the challenges faced with the application of linear programming technique to optimize disjunction of linear inequalities.

We have implemented our proposed techniques in a CPU-based prototypical analyzer named ARENA (Abstract Refinement Enhancer for Neural network verificAtion). In addition to verifying the robustness property of a network with respect to an image, ARENA is also capable of detecting adversarial examples that ascertain the falsification of the network property. We conducted experiments to assess the effectiveness of ARENA against the state-of-the-art tools, including the CPU-based verifiers DeepSRGR [10] and PRIMA [9], and the GPU-based verifier α, β-CROWN [13]. The results show conclusively that ARENA returns an average of 15.8% more conclusive images compared with DeepSRGR while terminates in comparable amount of time; and it also outperforms PRIMA by returning 16.6% more conclusive images. Furthermore, ARENA returns 14.8% more conclusive images than α, β-CROWN on average for selected networks.

We summarize our contributions below:

- ◇ We adapt the double description method proposed in the convex polytope domain [12] to solve disjuncts of constraints in Linear Programming (LP) encoding, allowing us to prune multiple adversarial labels together to increase overall efficiency.
- ◇ We leverage the multi-ReLU convex abstraction in PRIMA [9] to further refine the abstraction in the analysis process to increase verification precision.
- ◇ We utilize the solutions returned by the LP solver to detect adversarial examples and assert property violation when counter-examples are discovered.
- ◇ We conducted experiments comparing our prototypical analyzer ARENA against state-of-the-art verification tools, and demonstrate high effectiveness in our verification framework.

In the remaining part of the paper, we give an illustrative example showing the overall process of our method in Section 2, followed by a formal description of our methodologies in Section 3. We demonstrate our evaluation process and experimental results in Section 4. Section 5 discusses the current limitation, plan for future work and the generalization of our work. We give a literature review in Section 6, which contains closely related works with respect to our research scope. Finally, we summarize our work and conclude in Section 7.

2 Overview

In this section, we first describe the abstract refinement technique implemented in DeepSRGR [10]. Then, we discuss its limitations and introduce our approach to overcome them. Table 1 displays the notations we use throughout this section.

Π	the network constraint set/encoding
Υ	a potential adversarial region
P	the over-approximate convex hull

Table 1: Notations and descriptions of Section 2

2.1 Spurious Region Guided Refinement

DeepSRGR is a sound but incomplete verification method that relies on the polyhedral abstract domain in DeepPoly [7], where the abstract value of each network neuron x_i is designed to contain four elements (l_i, u_i, l_i^s, u_i^s). The *concrete* lower bound l_i and upper bound u_i pair forms a closed interval $[l_i, u_i]$ that over-approximates all the values that neuron x_i could take. The *symbolic* constraints l_i^s, u_i^s are linear expressions of x_i defined over preceding neurons with the requirement that $l_i^s \leq x_i \leq u_i^s$.

In the following, we illustrate the verification process where the abstract domain is used to verify the robustness property of a fully-connected network with ReLU activation (Figure 1) w.r.t the input space $I = [-1, 1] \times [-1, 1]$ of 2 input neurons x_1, x_2. This network has 3 output neurons y_1, y_2, y_3, corresponding to the three labels L_1, L_2, L_3 that an input in I can be classified as. Here, the robustness property which we aim to verify is that the neural network can always classify the entire input space I as label L_1, which corresponds to the output neuron y_1. More specifically, the verifier should be able to prove that the conditions $y_1 - y_2 > 0$ and $y_1 - y_3 > 0$ always hold for the entire input space I.

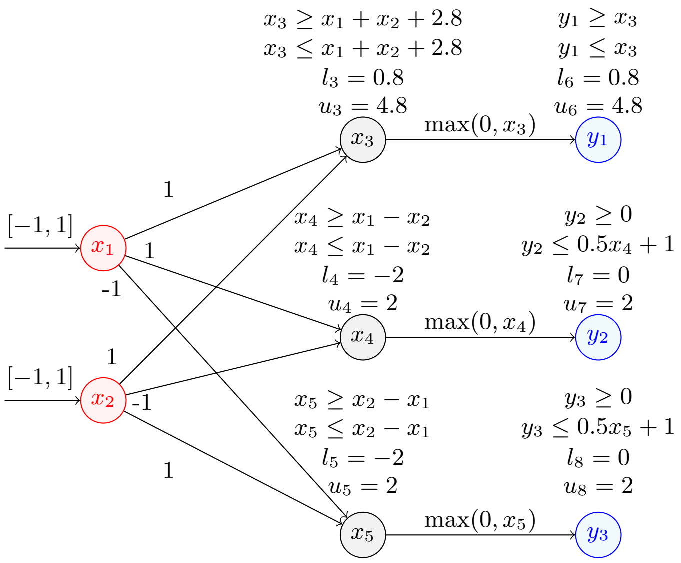


Fig. 1: The example network to perform DeepPoly abstract interpretation

Through the abstract interpretation technique, as deployed by DeepPoly, we can compute the abstract values for each neuron; these are displayed near the corresponding nodes. Specifically, the computed value for the lower bound of $y_1 - y_2$ and $y_1 - y_3$ are both -0.2 (the process of the lower bound computation is provided in Appendix A), which fails to assert that $y_1 - y_2 > 0$ and $y_1 - y_3 > 0$. In other word, DeepPoly cannot ascertain the robustness of the network for the given initial input space I. Given the over-approximation nature of abstract interpretation technique, it is not clear if the robustness property can be verified.

In order to further improve the robustness verification of the considered neural network, DeepSRGR conducts a *spurious region guided refinement* process that includes the following steps:

1. Obtain the conjunction of all linear inequities that encode the network, including the input constraint $x_1, x_2 \in [-1, 1]$ and the constraints within the abstract values of all neurons (*i.e.*, the constraints in Figure 1). We denote this network encoding constraint set as Π.

2. Take the conjunction of the current network encoding and the negation of the property to solve a potential *spurious* region. For example, the feasible region of $\Pi \wedge (y_1 - y_2 \leq 0)$ refers to a *potential* adversarial region (denote as Υ) that may contain a counterexample with adversarial label L_2 (corresponding to output neural y_2); whereas the region *outside* of Υ is already a safe region that will not be wrongly classified as L_2. However, the region Υ may exist only due to the over-approximate abstraction but does not contain any true counterexample. Therefore, this region is spuriously constructed and could be eliminated. If we successfully eliminate Υ, then we can conclude that label L_2 will not be a valid adversarial label since y_2 never dominates over y_1.
3. To eliminate the region Υ, DeepSRGR uses the constraints of the region to refine the abstraction using linear programming (LP). For instance, we take Π and $y_1 - y_2 \leq 0$ as the constraint set of linear programming. To obtain tighter bounds for input neurons and unstable ReLU neurons[1], we set the objective function of LP as $\min(x_i)$ and $\max(x_i)$ where $i \in [1, 2, 4]$. The new solved intervals are highlighted in red in Figure 2, where all the current neuron intervals now specify the region Υ.

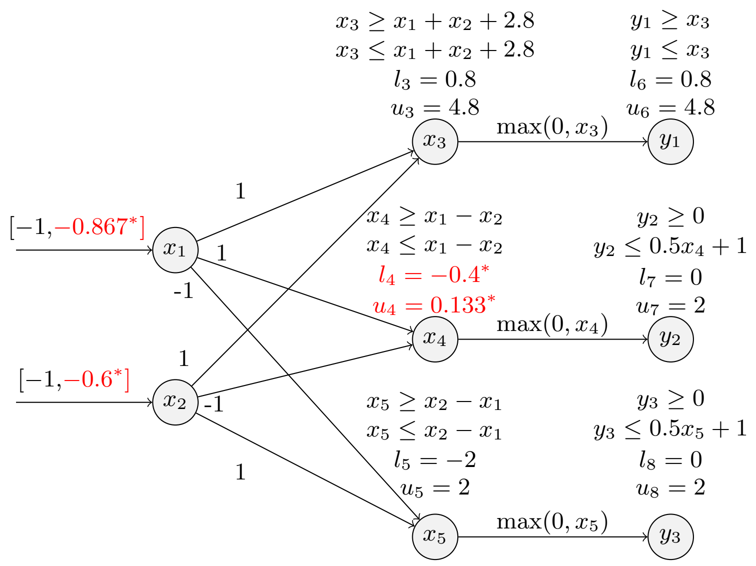


Fig. 2: The effect of applying LP-based interval refinement (in red marked by *)

4. DeepSRGR leverages those tighter bounds to guide the abstract interpretation of the region Υ in the next iteration. It performs a second run on DeepPoly and makes sure that this second run compulsorily follows the new bounds computed in the previous step. As shown in Figure 3, the blue colored part refers to the updated abstract values during the second execution of DeepPoly, where the abstraction of all neurons are refined due to the

[1] Unstable ReLU neuron refers to a ReLU neuron whose input range can be both negative and positive (like y_2, y_3).

tighter bounds (red colored part) returned by LP solving. Now the lower bound of $y_1 - y_2$ is 0.7, making $y_1 - y_2 \leq 0$ actually infeasible within the region Υ. Therefore, we conclude that Υ is a spurious region that does not contain any true counterexample, and we can eliminate adversarial label L_2.

5. If we fail to detect $y_1 - y_2 \leq 0$ to be infeasible, DeepSRGR iterates the process from step 2-4 where it calls LP solving and re-executes DeepPoly on the new bounds until it achieves one of the termination conditions: (i) It reaches the maximum number of iterations (DeepSRGR sets it to be 5 by default); or (ii) it detects infeasibility for the spurious region.

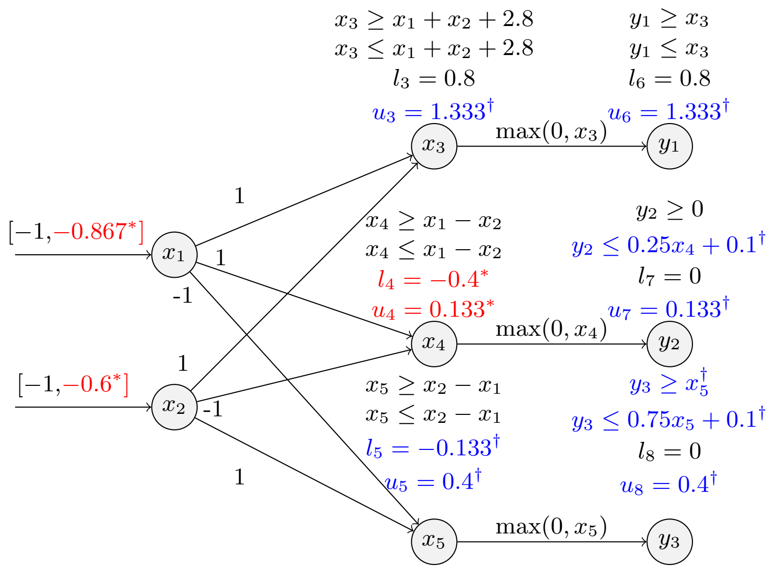


Fig. 3: Results of the second run of DeepPoly (in blue marked by †)

Similarly, after eliminating the adversarial label L_2, DeepSRGR will apply the same process to eliminate the spurious region defined by $\Pi \wedge (y_1 - y_3 \leq 0)$, which corresponds to the output neural y_3 and the adversarial label L_3.

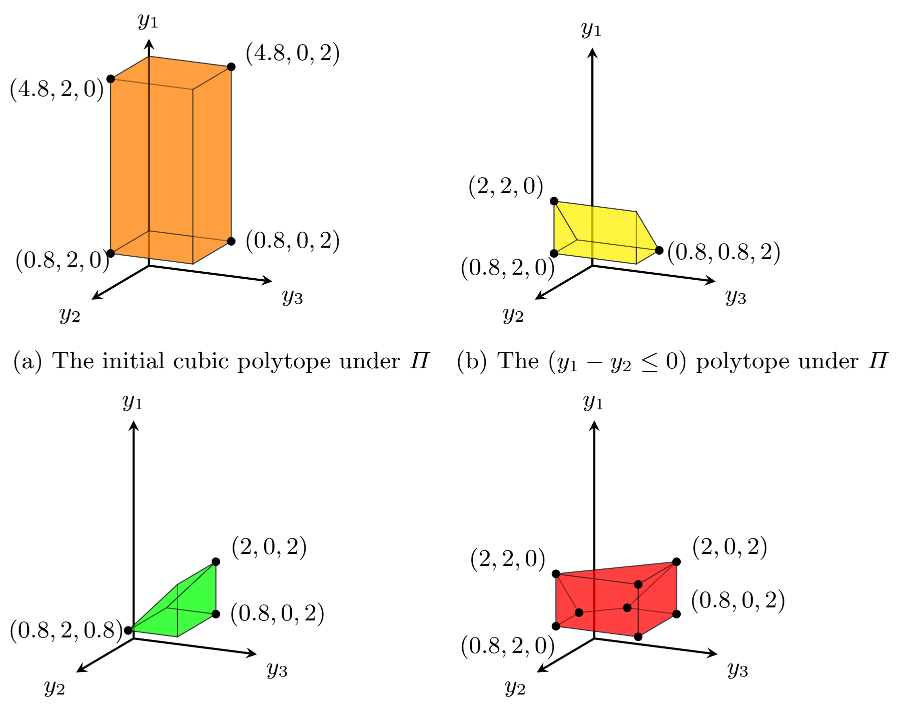
In summary, DeepSRGR uses iterative LP solving and DeepPoly execution to attempt to eliminate spurious regions which do not contain counterexamples. Assuming the ground-truth label to be L_c, DeepSRGR runs this refinement process for each region $\Pi \wedge (y_c - y_t \leq 0)$ where $t \neq c$. If DeepSRGR is able to eliminate all adversarial labels related to output neurons y_t where $t \neq c$, then it successfully ascertains the robustness property of the image. If DeepSRGR fails to eliminate one of the adversarial labels within the iteration boundary, the robustness result remains inconclusive.

2.2 Scaling up with multiple adversarial label elimination

We mention three contributions in Section 1 including efficiency improvement, precision improvement and adversarial example detection. In this section, we only give an overview of our multiple adversarial label elimination method which aims to improve the analysis efficiency; we defer the discussion of the remaining part of our system to Section 3.

As mentioned in Section 2.1, DeepSRGR invokes the refinement process to sequentially eliminate each spurious region $\Pi \wedge (y_c - y_t \leq 0)$, which corresponds to the adversarial label L_t $(t \neq c)$. For an n-label network, it requires $n-1$ refinement invocations in the worst case, with each invocation taking possibly several iterations. To speed up the analysis, we eliminate multiple spurious regions at the same time in one refinement process.

For example, we aim to detect infeasibility in $\Pi \wedge ((y_1 - y_2 \leq 0) \vee (y_1 - y_3 \leq 0))$ so as to eliminate both adversarial labels L_2 and L_3 simultaneously. The technical challenge behind this *multiple adversarial label* elimination is that linear programming does not naturally support the *disjunction* of linear inequalities. To address this challenge, we compute the over-approximate convex hull P of $(y_1 - y_2 \leq 0) \vee (y_1 - y_3 \leq 0)$ under network encoding Π. As P will be represented as a set of linear inequalities, linear programming is amenable to handle $\Pi \wedge P$.



(a) The initial cubic polytope under Π (b) The $(y_1 - y_2 \leq 0)$ polytope under Π

(c) The $(y_1 - y_3 \leq 0)$ polytope under Π (d) The convex hull of union of (b),(c)

Fig. 4: The convex polytopes under network encoding, with respect to (y_1, y_2, y_3).

In detail, the initial convex polytope associated with y_1, y_2, y_3 is a 3-D cube pictured in Figure 4a, where $y_1 \in [0.8, 4.8], y_2 \in [0, 2], y_3 \in [0, 2]$ after we perform

DeepPoly as shown in Figure 1. The convex polytope for the constraint $y_1 - y_2 \leq 0$ under the network encoding Π corresponds to the shape in Figure 4b where $y_1 - y_2 \leq 0$ is a cutting-plane imposed on the initial cube in Figure 4a. Similarly, the projection of $y_1 - y_3 \leq 0$ to a convex polytope can be visualized in Figure 4c. We further compute the over-approximate convex hull P of the union of the two polytopes as in Figure 4d.

We can observe that P is defined by 8 vertices (annotated as eight black extreme points). It is worth-noting that these 8 vertices actually come from either vertices in Figure 4b or vertices in Figure 4c. We will provide the explanation and the theory on how to compute the convex hull of the union of two polytopes in Section 3.2. Explicitly, P can also be represented by the following constraint set (1), which correspond to the 7 red-colored surfaces in Figure 4d:

$$\begin{array}{cccc} -y_1 + y_2 + y_3 \geq 0 & y_2 \geq 0 & y_3 \geq 0 & -1 + 1.25y_1 \geq 0 \\ 2 - y_1 \geq 0 & 2 - y_2 \geq 0 & 2 - y_3 \geq 0 & \end{array} \tag{1}$$

We take the network encoding Π and constraint set of P as the input to the LP solver, and conduct interval solving as in Section 2.1. We annotate the new bounds obtained through LP solving as red color, and the updated abstract values after the second abstract interpretation as blue color in Figure 5. The lower bounds of both $y_1 - y_2$ and $y_1 - y_3$ now become 0.2, making it infeasible to achieve $y_1 - y_2 \leq 0$ or $y_1 - y_3 \leq 0$. Therefore, we successfully do the verification with just one refinement process invocation.

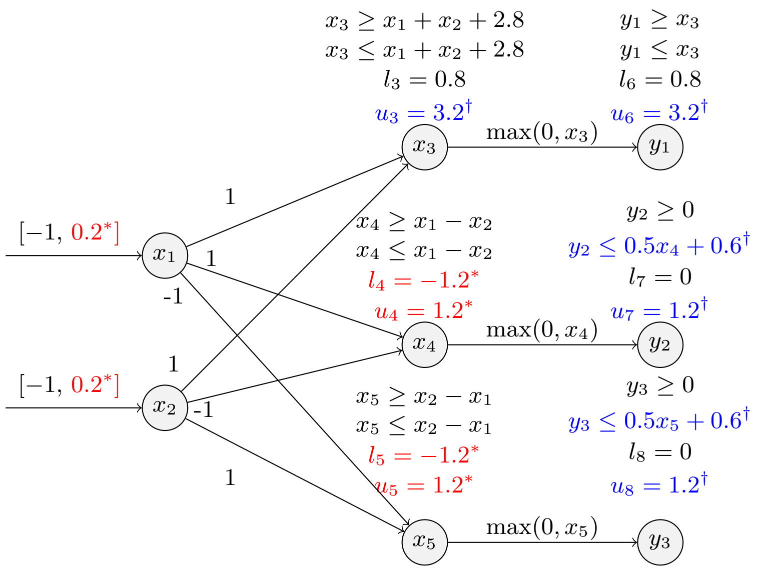


Fig. 5: The new intervals (in red with *) with two-adversarial labels encoding and new abstraction introduced by the second run of DeepPoly (in blue with †)

3 Methodologies

As described in Section 2, we identify the feasible region of the network encoding and the negation of a property (i.e. $\Pi \wedge (y_c - y_t \leq 0) : t \neq c$) as a potential spurious region and leverage the refinement process to ascertain and possibly eliminate such spurious regions. To further improve the precision and efficiency, we propose three techniques as we have summarized in Section 1:

1. We update the negation of the property encoding to capture multiple spurious regions at the same time, as we demonstrate the example on $(y_1 - y_2 \leq 0) \vee (y_1 - y_3 \leq 0)$ in Section 2.2. This method allows us to reuse the linear programming part among several spurious regions and improve efficiency.
2. We leverage the multi-ReLU convex abstraction proposed in PRIMA [9] to obtain a more precise network encoding Π, which helps to increase the verification precision.
3. We detect adversarial examples to falsify robustness property. In particular, as the LP solver finds the conjunction of the network encoding and the negation of the property to be feasible, its optimization solution could actually ascertain a property violation and help us conclude with falsification.

We will discuss the three methodologies in separate subsections, and conclude this section with an overall description of our verification framework ARENA. Table 2 shows the notations we use in the main text of this section.

Π	the network constraint set/encoding
Ω	the multi-ReLU constraint set
Λ	the involved variable set during convex computation
Θ	the initial multidimensional octahedra during convex computation
P_i^H	the convex polytope in H-representation
P_i^V	the convex polytope in V-representation

Table 2: Notations and descriptions of Section 3

3.1 Multi-ReLU network encoding

As mentioned before, the constraint set subject to linear programming resolution is a conjunction of the network encoding and the negation of the property. In this subsection, we describe our network constraint construction. In the next subsection, we will describe the encoding of the negated property. Particularly, we capture the dependencies between the ReLU neurons in the same layer in our network encoding by leveraging the multi-ReLU convex relaxation in PRIMA.

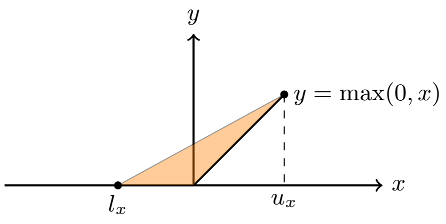


Fig. 6: The triangle approximation of a single ReLU neuron

As depicted in Figure 6, DeepSRGR uses a triangular shape to encode each ReLU neuron independently, where the ReLU node $y = \max(0, x)$ with $x \in [l_x, u_x]$. The triangular shape is defined by three linear constraints:

$$y \geq x \qquad y \geq 0 \qquad y \leq \frac{u_x}{u_x - l_x}(x - l_x)$$

This looseness of ReLU encoding can inhibit precision improvement in DeepSRGR. As a matter of fact, it has been reported that, when they increase the maximum number of iterations from 5 to 20, only two more properties can be verified additionally, and no more properties can be verified when they further increase from 20 to 50 [10].

To break this precision barrier, we deploy the technique of *multi-ReLU relaxation* in PRIMA [9] where they compute the convex abstraction of k-ReLU neurons via novel convex hull approximation algorithms. For instance, if $k = 2$ and the ReLU neurons in the same layer are denoted by y_1, y_2, and the inputs to these two ReLU neurons are x_1, x_2 respectively, PRIMA will compute a convex hull in (y_1, y_2, x_1, x_2) space to capture the relationship between the two ReLU neurons and their inputs. An example of the convex hull is defined as:

$$\begin{aligned}\Omega = \{\, & x_1 + x_2 - 2y_1 - 2y_2 \geq -2, \quad 0.375x_2 - y_2 \geq -0.75, \\ & -x_1 + y_1 \geq 0, \quad -x_2 + y_2 \geq 0, \quad y_1 \geq 0, \quad y_2 \geq 0 \,\}\end{aligned}$$

As we can see, Ω contains the constraint $x_1 + x_2 - 2y_1 - 2y_2 \geq -2$ that correlates (y_1, y_2, x_1, x_2) all together, which is beyond the single ReLU encoding. In general, PRIMA splits the input region into multiple sub-regions and then computes the convex hull of multiple ReLU neurons. For example, splitting the input region along $x_1 = 0$ results in two sub-regions where $y_1 = x_1$ (y_1 is activated) and $y_1 = 0$ (y_1 is deactivated). In each sub-region, the behavior of y_1 is determinate and this yields a tighter or even exact convex approximation. Finally, PRIMA computes a joint convex over-approximation (as in Ω) of the convex polytopes computed for each sub-region.

For deployment, we consider 3-ReLU neurons in our paper. We filter out the unstable ReLU neurons in each ReLU layer, and divide them into a set of 3-ReLU groups with one overlapping neuron between two adjacent groups as shown in Figure 7, where a dashed box identifies a 3-ReLU group. We then leverage PRIMA to compute the constraints for each 3-ReLU group, and add those additional constraints into the original network encoding in order to obtain a more precise network abstraction and better verification precision.

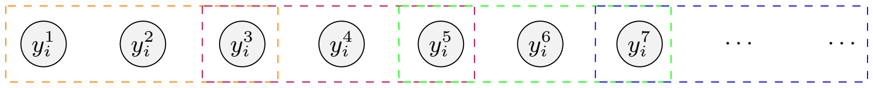


Fig. 7: The 3-ReLU grouping for unstable ReLU neurons in the same layer i, where we use PRIMA to compute the convex relaxation for each group.

3.2 Multiple adversarial label elimination

We now explain how we encode the negated property, especially when we take multiple spurious regions into consideration. As demonstrated in Section 2.2, to make it amenable for LP encoding, we need to compute the over-approximate convex hull of the union of multiple convex polytopes like in Figure 4d. To explain the theory behind, we first introduce the required knowledge with respect to convex polytope representation. The convex polytope in this paper refers to a bounded convex polytope that is also a convex region contained in the $n-$dimensional Euclidean space R^n. There are two essential definitions of a convex polytope: as the intersection of half-space constraints (H-representation) and as the convex hull of a set of extremal vertices (V-representation) [14].

H-representation. A convex polytope can be defined as the intersection of a finite number of closed half-spaces. A closed half-space in an n-dimensional space can be expressed by a linear inequality:

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n \leq b \tag{2}$$

A closed convex polytope can be taken as the set of solutions to a linear constraint set, just like the constraint set (1) shown in Section 2.2.

V-representation. A closed convex polytope can also be defined as the convex hull with a finite number of points where this finite set must contain the set of extreme points of the polytope (i.e. the black-colored dots in Figure 4d).

Double description. The Double description method [12] aims to maintain both V-representation and H-representation during computation. This "duplication" is beneficial because to compute the intersection of two polytopes in H-representation is trivial since we only need to take the union set of the half-space constraints. On the other hand, to compute the convex hull of the union of two polytopes is trivial in V-representation as we take the union set of the vertices. The program `cddlib`[2] is an efficient implementation of the double description method, which provides functionalities that enable transformation from V-representation to H-representation (named as convex hull problem); and vice versa (named as vertex enumeration problem).

We leverage this V-H transformation in cddlib to compute the convex hull of the union of multiple convex polytopes. We set up a *batch size* δ to be in $[2, 5]$[3],

[2] https://github.com/cddlib/cddlib
[3] We explain our parameter range setting in Section 5 and also provide the batch size study experiments in Section 4.4.

which defines the number of spurious regions to be considered simultaneously. Assume that the ground truth label is L_c, the related adversarial labels are $L_1, \cdots, L_\delta$, the convex-hull computation of $(y_c - y_1 \leq 0) \vee \cdots \vee (y_c - y_\delta \leq 0)$ is conducted as follows:

Polytope computation for each spurious region. We compute the H-representation of the polytope for each spurious region in the $(\delta+1)$-dimensional space with respect to the variable set $\Lambda = y_c, y_1, \cdots, y_\delta$. Intuitively, we obtain the H-representation of polytope $(y_c - y_i \leq 0)$ by taking the interval constraints of Λ (which is a multidimensional cube) conjunct with $y_c - y_i \leq 0$, as our example in Figure 4. But this encoding is coarse as we neglect the dependencies between Λ that are in the same layer. For a more precise encoding, we follow the idea of [8] and compute the multidimensional octahedra Θ of $y_c, y_1, \cdots, y_\delta$, which yields $3^{\delta+1} - 1$ constraints defined over Λ. Therefore, the H-representation of polytope $(y_c - y_i \leq 0)$ will be the constraint set Θ and $y_c - y_i \leq 0$.

Union of convex polytopes. We obtain the H-representation of the δ polytopes in the previous step and denote them by $P_1^H, \cdots, P_\delta^H$ respectively. Since the union of polytopes is trivial in V-representation – as mentioned earlier, we use the H-V transformation in cddlib to generate these V-representations of the δ polytopes (referred to as $P_1^V, \cdots, P_\delta^V$). As illustrated in Figure 8, we then produce the union set P_u^V of these vertices sets and transform it to its H-representation P_u^H, which is the convex hull of the union of δ polytopes. As P_u^H is represented by a set of linear inequalities, we conjunct it with the network encoding Π and submit the constraints for LP solving.

$$\begin{tikzcd}
P_1^H \arrow[rr, "\text{HtoV}"] & & P_1^V \arrow[dr, "\cup"] & & \\
\vdots & \vdots & \vdots \arrow[r, "\cup"] & P_u^V \arrow[r, "\text{VtoH}"] & P_u^H \\
P_\delta^H \arrow[rr, "\text{HtoV}"] & & P_\delta^V \arrow[ur, "\cup"'] & &
\end{tikzcd}$$

Fig. 8: The convex hull computation of the union of δ convex polytopes

3.3 Adversarial example detection

As mentioned previously, we take the conjunction of the network encoding and the negation of the property as the input constraint set to the LP solver and aim to eliminate spurious region(s) when detecting infeasibility. On the other hand, when the constraint set is feasible, we can set the input neurons and the unstable ReLU neurons as objective function and try to resolve for tighter intervals. In fact, a feasible constraint set indicates the possibility of a property violation. The LP solver not only returns the optimized value of the objective function, it also returns a solution that leads to the optimization, which could be a potential counter-example of robustness. Therefore, we include a supplementary procedure that takes each optimal solution obtained from the LP solver and checks if it constitutes an adversarial example.

This process brings forth two benefits: (1) it detects counter-examples and asserts the violation of robustness; (2) it enables the process to *terminate early with falsification* (once a counter-example is discovered) instead of exhausting all the iterations.

3.4 The Verification Framework ARENA

We now present an overview of our verification framework ARENA. In addition to the implementation of the three main technical points covered earlier, our framework includes the following optimizations as well.

Algorithm 1: Overall analysis procedure in ARENA

Input:
- N: input neural network with input layer γ_{in}, and output neurons $y_1, ..., y_n$
- y_c: the output neuron corresponding to the ground truth label L_c $(1 \leq c \leq n)$
- δ is the refinement batch size (the number of adversarial labels to be eliminated by batch in each iteration).

Output: Verification result (Verified for robustness verified, Falsified for robustness violated, Inconclusive for inconclusive result).

```
1: $(res, A_N) \leftarrow$ VerifyByDeepPoly($N$)          // Result and network abstraction
2: if $res =$ Verified then
3:     return Verified
4: else
5:     $\Pi \leftarrow$ GetConstraintsInNetwork($N$, $A_N$)
6:     $\mathcal{L}_{adv} \leftarrow \{L_i \mid$IsFeasible$(\Pi \wedge y_c - y_i \leq 0), \forall i \neq c\}$   // All adversarial labels
7:     $\mathcal{L}_{adv} \leftarrow$ SortByEstimatedSpuriousRegionSize($\mathcal{L}_{adv}$)      // Sort decreasingly
8:     $\mathcal{L}_{elim} \leftarrow \varnothing$; $i \leftarrow 0$; $iter\_num \leftarrow 999$      // Initialization
9:     while $i <$ Length($\mathcal{L}_{adv}$) do
10:        if $iter\_num > 2$ then
11:            $status, iter\_num \leftarrow$ RefineWithKReLU($N, \Pi, L_c, \mathcal{L}_{adv}[i], \mathcal{L}_{elim}$)
12:            if $status =$ Falsified or $status =$ Inconclusive then
13:                return $status$
14:            else                                  // $status =$ Verified
15:                $\mathcal{L}_{elim} \leftarrow \mathcal{L}_{elim} \cup \{\mathcal{L}_{adv}[i]\}$
16:                $i \leftarrow i + 1$
17:        else
18:            $\mathcal{L}' \leftarrow$ GetNextAdversarialLabelBatch($\mathcal{L}_{adv}, \delta$)
19:            $status \leftarrow$ EliminateAdversarialLabels($N, \Pi, L_c, \mathcal{L}', \mathcal{L}_{elim}$)
20:            if $status =$ Falsified or $status =$ Inconclusive then
21:                return $status$
22:            else                                  // $status =$ Verified
23:                $\mathcal{L}_{elim} \leftarrow \mathcal{L}_{elim} \cup \{\mathcal{L}'\}$
24:                $i \leftarrow i +$ SizeOf($\mathcal{L}'$)
25:    if $\mathcal{L}_{elim} = \mathcal{L}_{adv}$ then
26:        return Verified
27:    else
28:        return Inconclusive
```

Optimization 1: *Prioritising elimination of larger spurious regions.* We choose to order the sequence of the spurious regions according to the descending order of the respective regions' sizes, by eliminating the "toughest" spurious region first. Since robustness only holds when all spurious regions are eliminated, we terminate the refinement process early if we fail to prune a larger spurious region. As it is difficult to compute the actual size of the spurious region, we deploy the metric in DeepSRGR where they take the lower bound of expression $y_c - y_i$ given by DeepPoly as the estimation of the region size, i.e. the smaller this value is, the larger the region is likely to be and thus it would be tougher for us to eliminate the region.

Optimization 2: *Cascading refinement.* Our system is designed to apply increasingly more scalable and less precise refinement methods. We hereby define process RefineWithKReLU as the refinement method with multi-ReLU encoding for the network and multi-adversarial label pruning feature *disabled*. Similarly, process EliminateAdversarialLabels(δ) is the refinement method with multi-ReLU encoding, and *taking into consideration* δ spurious regions *simultaneously*. With additional over-approximation error potentially being introduced by computing the union of polytopes, EliminateAdversarialLabels(δ) is less precise method compared to RefineWithKReLU but more scalable as it eliminates δ spurious regions simultaneously. We first use RefineWithKReLU to eliminate the larger spurious regions and record the number of iterations ς required to prune the current spurious region. If $\varsigma \leq 2$, this indicates that it is rather amiable to prune the current spurious region, and affordable to call upon EliminateAdversarialLabels(δ) to eliminate the remaining smaller spurious regions.

We present the overall analysis procedure in Algorithm 1. To begin with, we only apply the refinement process to images that fail to be verified by DeepPoly (lines 4). For refinement, we first obtain all the network constraints generated during abstract interpretation (line 5) and all potential adversarial labels (line 6). Then we call upon the processes RefineWithKReLU (line 11) or EliminateAdversarialLabels(δ) (line 19) to eliminate one or multiple spurious regions as stated in optimization 2 mechanism. The analyzer returns "Falsified" value if it detects an adversarial example (lines 13, 21); or it returns "Inconclusive" value if it fails to eliminate one of the adversarial labels and fails to find a counter-example (lines 13, 21, 28). We declare verification to be successful if and only if we can eliminate all the adversarial labels (lines 25-26).

Details of the two refinement processes are presented in Algorithms 2 and 3 (in Appendix B). These two algorithms only differ in the property encoding (line 3-4 in Algorithm 3 vs lines 3-4 in Algorithm 2). Reading Algorithm 2 more closely: it first computes the convex hull of the union of the spurious regions according to Section 3.2 (line 3). Next, it conjuncts the convex hull with the network encoding in line 4, and add constraint $y_c - y_t > 0$ for each of the previously eliminated adversarial label L_t (lines 5-6); this helps to reduce the solution space further. If the combined constraint set Σ is found to be infeasible, the process returns "verified" since violation of the property cannot be attained (lines 7-8). But if Σ is feasible, the process leverages it to further tighten the bounds for input and unstable ReLU neurons and updates the network (lines 9, 14). Moreover, the process checks if each LP solution is a valid counter-example; if so, it returns "Falsified" (lines 10-11, 15-16). With the newly solved bounds, the process then re-runs DeepPoly to obtain a tighter network encoding (lines 17-18) that is more amenable to encounter infeasibility in the latter iterations. Finally, the process returns "inconclusive" if it fails to conclude within the maximum number of iterations (line 20).

Algorithm 2: The refinement procedure EliminateAdversarialLabels

```
Function Name: EliminateAdversarialLabels($N, \Pi, L_c, \mathcal{L}', \mathcal{L}_{elim}$)
Input:
  – $N$: input neural network with input layer $\gamma_{in}$, and output neurons $y_1, ..., y_n$
  – $\Pi$: the constraint set of $N$
  – $y_c$: the output neuron corresponding to the ground truth label $L_c$ $(1 \leq c \leq n)$
  – $\mathcal{L}'$, $\mathcal{L}_{elim}$: the batch of adversarial labels to be refined, and the list of previously
    eliminated labels
Output: the refinement status
   $counter = 0$
   while $counter < \tau$ do                              // $\tau$ is an iteration threshold
      $P_u^H \leftarrow$ ComputeConvexHull($N, \mathcal{L}'$)     // $P_u^H$ is the convex hull of polytopes
      $\Sigma \leftarrow \Pi \wedge P_u^H$                                   // Initialize constraint set
      for all $L_t \in \mathcal{L}_{elim}$ do
         $\Sigma \leftarrow \Sigma \wedge (y_c - y_t > 0)$
      if IsInfeasible($\Sigma$) then
         return Verified
      $N \leftarrow$ LPSolveInputInterval($\Sigma, \gamma_{in}$)      // Update network with new bounds
      if ExistsAnAdversarialExample($N$) then
         return Falsified
      for all ReLU layer $\gamma'_k$ in $N$ do
         $\gamma_k \leftarrow$ GetPrecedingInputAffineLayer($\gamma'_k$)
         $N \leftarrow$ LPSolveUnstableReLUs($\Sigma, \gamma_k$)                  // Update new bounds
         if ExistsAnAdversarialExample($N$) then
            return Falsified
      $A \leftarrow$ RecomputeNetworkAbstractionByDeepPoly($N$)
      $\Pi \leftarrow$ GetConstraintsInNetwork($N, A$)
      $counter = counter + 1$
   return Inconclusive
```

4 Experiments

We implemented our method in a prototypical verifier called ARENA in both C and C++ programming languages (C++ is used for the k-ReLU computation feature, while the rest of the system is implemented in C). Our verifier is built on top of DeepPoly in [15]: it utilizes DeepPoly as the back-end abstract interpreter for neural networks. Moreover, it uses `Gurobi`[4] version 9.5 as the LP solver for constraints generated during abstract refinement.

We evaluate the performance of ARENA with state-of-the-art CPU-based verifiers including DeepSRGR [10], PRIMA [9], and DeepPoly [7]. Furthermore, we compare with α, β-CROWN [13], which is GPU-based and the winning tool of VNN-COMP 2022 [16]. The evaluation machine is equipped with two 2.40GHz Intel(R) Xeon(R) Silver 4210R CPUs with 384 GB of main memory and a NVIDIA RTX A5000 GPU. The implementation is 64-bit based.

Note that DeepSRGR [10] was purely implemented in Python, while the main analysis in ARENA, PRIMA and DeepPoly were implemented in C/C++. Furthermore, this original version of DeepSRGR does not support convolutional networks nor the ONNX network format in our benchmark. Therefore, to avoid any runtime discrepancy introduced by different languages and to support our tested networks, we re-implemented the refinement technique of DeepSRGR in C, and conducted the experiment on the re-implemented DeepSRGR, where we release our re-implementation of DeepSRGR at this link: `https://github.com/arena-verifier/DeepSRGR`. The source code of our verifier ARENA is available online at: `https://github.com/arena-verifier/ARENA`.

4.1 Experiment setup

Evaluation datasets and testing networks. We chose the commonly used MNIST [17] and CIFAR10 [18] datasets. MNIST is an image dataset with hand-written digits, containing gray-scale images with 28×28 pixels. CIFAR10 includes RGB three-channel images with size 32×32. Our testing image set consists of the first 100 images of the test set of each dataset, which is accessible from [15].

We selected fully-connected (abbreviated as FC) and convolutional (abbreviated as Conv) networks from [15] as displayed in Table 3, with up to around 50k neurons. We explicitly list the number of hidden neurons, the number of activation layers, trained defense[5], and the number of candidate images for each network. Here, the candidate images refer to those testing images that can be correctly classified by the network and we only apply robustness verification on the candidate images.

Robustness analysis. We conducted robustness analysis against L_∞ norm attack [19] with a perturbation parameter ϵ. Assuming that each pixel in the test image originally takes an intensity value p_i, it now takes an intensity interval $[p_i - \epsilon, p_i + \epsilon]$ after applying L_∞ norm attack with a specified constant ϵ.

This naturally forms an input space defined by $\bigtimes_{i=1}^{n}[p_i - \epsilon, p_i + \epsilon]$, and all the tools attempt to verify if all the "perturbed" images within the input space will be classified the same as the original image by the tested network. If so, we claim that the robustness property is verified. On the contrary, if we detect a

[4] `https://www.gurobi.com/`

[5] A trained defense refers to a defense method against adversarial samples, with the purpose of improving the robustness property of the network

Neural Net	**ARENA**			**DeepSRGR**		**PRIMA**		**DeepPoly**	
	Verify	Falsify	Time	Verify	Time	Verify	Time	Verify	Time
M_3_100	63	5	87.65	54	68.76	69	123.73	24	0.105
M_5_100	77	7	250.39	67	153.75	53	19.15	25	0.522
M_6_100	45	6	650.10	38	324.14	38	173.03	23	0.280
M_9_100	44	10	1527.2	34	1004.4	34	191.60	30	0.587
M_6_200	48	3	1514.2	35	1312.3	34	222.45	25	0.313
M_9_200	43	6	3857.8	35	3536.7	29	238.63	29	0.536
M_convSmall	69	7	176.93	66	251.27	70	84.23	31	0.605
M_convMed	66	5	2054.9	60	2826.6	59	125.88	24	1.646
C_6_500	31	9	2703.3	24	3985.2	20	269.96	16	12.22
C_convMed	31	7	3417.1	30	4385.4	30	230.74	21	3.87

Table 4: The number of verified/falsified images and average execution time (in seconds) per image for MNIST and CIFAR10 network experiments

counter-example with a different classification label, we assert the falsification of the robustness property. Finally, if we fail to conclude with verification or falsification, we return *unknown* to the user, meaning that the analysis result is inconclusive. We set up a challenging perturbation ϵ for each network and show in Table 3.

Network	**Dataset**	**Type**	ϵ	**#Layer**	**#Neurons**	**Defense**	**Candidates**
M_3_100	MNIST	FC	0.028	3	210	None	98
M_5_100	MNIST	FC	0.08	6	510	DiffAI	98
M_6_100	MNIST	FC	0.025	6	510	None	99
M_9_100	MNIST	FC	0.023	9	810	None	97
M_6_200	MNIST	FC	0.016	6	1,010	None	99
M_9_200	MNIST	FC	0.015	9	1,610	None	97
M_convSmall	MNIST	Conv	0.11	3	3,604	None	100
M_convMed	MNIST	Conv	0.1	3	5,704	None	100
M_convBig	MNIST	Conv	0.306	6	48,064	DiffAI [20]	95
C_6_500	CIFAR10	FC	0.0032	6	3,000	None	56
C_convMed	CIFAR10	Conv	0.006	3	7,144	None	67

Table 3: Experimental Fully Connected and Convolutional Networks

4.2 Comparison with the CPU-based verifiers

We present the robustness analysis results for ten networks in Table 3 and we describe the parameter configurations of our tool ARENA in Appendix C. To execute PRIMA, we use the "refinepoly" domain in ERAN [15] and the parameter setting of PRIMA are given in Appendix E.

We report the experiment results drawn from different tools for MNIST and CIFAR10 networks in Table 4. For ARENA, we report the number of verified images, the number of falsified images and the average execution time for each testing image. DeepSRGR does not detect adversarial examples, neither does it attempt to assert violation of the property. PRIMA, on the other hand, returns two unsafe image for one MNIST network only (the detailed results are given in Appendix E). Thus we omit the falsification column from the report for these two methods. Due to time limitation, for networks M_9_200, C_6_500 and C_convMed, we set a 2 hours timeout for each image. If the refinement process fails to terminate before timeout, we consider the verification as inconclusive.

We observe from Table 4 that ARENA returns significantly more conclusive images (including both verified and falsified images) for all the networks than DeepSRGR, with comparable or even less execution time than that of DeepSRGR. ARENA also returns more conclusive images for all the networks than PRIMA, except for the subject MNIST_3_100, where ARENA returns less verified images than PRIMA. Our in-depth investigation reveals that it is because two out of the three hidden layers have their ReLU neurons being encoded *exactly* with MILP in PRIMA.

These analysis results are better visualized in Figure 9 and Figure 10 in Appendix D. As can be seen in Appendix D, ARENA generally returns more conclusive images than the rest of the tools. On average, ARENA returns 15.8% more conclusive images than DeepSRGR and 16.6% more conclusive images than PRIMA respectively for the testing networks.

4.3 Comparison with the GPU-based verifier α,β-CROWN

Furthermore, we selected five networks from Table 3 to compare with the state-of-the-art tool α,β-CROWN (alpha-beta-CROWN) [13], which is a complete verification tool in the sense that it will produce a conclusive answer given infinite amount of time. The experiment configuration of α,β-CROWN is provided in Appendix F and we report in detail with the average execution time for all tested images, average execution time for those verified images and average execution time for those falsified images.

Neural Net	**ARENA**					**α,β-CROWN**				
	Verify		**Falsify**		**Ave**	**Verify**		**Falsify**		**Ave**
	Num	**Time**	**Num**	**Time**	**Time**	**Num**	**Time**	**Num**	**Time**	**Time**
M_3_100	63	71.57	5	27.88	87.65	54	46.29	11	2.804	25.15
M_5_100	77	211.7	7	72.47	250.3	53	91.41	10	4.40	48.67
M_convSmall	69	109.4	7	78.49	176.9	39	6.55	16	3.21	3.06
M_convMed	66	1353.4	5	658.7	1625.9	30	8.08	18	2.77	2.90
M_convBig	53	226.1	30	553.8	589.52	49	6.25	24	4.77	4.21

Table 5: The number of verified/falsified images and detailed execution time for ARENA and α,β-CROWN, time is presented in *seconds*

As can be seen from Table 5, α, β-CROWN runs significantly faster than our tool ARENA due to GPU acceleration. On the other hand, our tool is capable of identifying significantly more verified images than α, β-CROWN due to its refinement process. α, β-CROWN falsifies more images for most of the tested networks as it deploys existing PGD attack[6] method [21]. However, ARENA verifies more images than α, β-CROWN in all experimented networks. On average, ARENA manages to return 14.8% more conclusive (including verified and falsified) images. The result shows promisingly that ARENA, despite being based on an incomplete technique, can still obtain more precise results in practice than the complete method of α, β-CROWN.

4.4 Multi-adversarial label parameter study

In this experiment, we selected three networks from Table 3 to assess how the batch size parameter δ may impact the verification precision and execution time.

Theoretically, a larger value of δ may lead to a more efficient analysis process as it allows more adversarial regions to be eliminated at the same time. However, setting the parameter to be δ requires the computation of the union of δ convex polytopes. This in turn may introduce more over-approximation error and may jeopardize the analysis precision. As δ aims to speed up the refinement process, we present the number of images that are verified through iterative refinement process and the average verification time for *those refined images*[7]. The experimental results are shown in Table 6 where we compare among parameters $\delta = 2, 3, 4, 5$ and with multi-adversarial label feature being disabled (the same as setting $\delta = 1$).

Network	**ARENA**									
	(disabled)		(δ=2)		(δ=3)		(δ=4)		(δ=5)	
	VTR	Time(s)	VTR	Time(s)	VTR	Time(s)	VTR	Time(s)	VTR	Time(s)
M_3_100	41	142.91	39	144.67	39	135.62	39	124.99	38	127.77
M_6_100	22	1414.4	22	1312.6	22	1250.5	22	1202.4	22	1047.9
M_6_200	26	4809.7	23	2552.2	23	1828.2	23	1663.2	23	1297.0

Table 6: The number of **v**erified images **t**hrough the **r**efinement process (VTR) and average verification time per refined image for different δ setting.

The experiment results show that the choice of a larger δ still allows us to achieve closely comparable precision while requires less execution time. Since an

[6] Projected Gradient Descent (PGD) attack is a white-box attack with access to the model gradients, and it leverages the gradients to detect the counter-example through a constrained optimisation problem.

[7] As we only apply the refinement process to those testing images that DeepPoly fails to verify, the "refined" images refers to those images that are successfully verified through our refinement process, NOT through the original DeepPoly process.

appropriate set-up of parameters leads to a better combination of precision and efficiency, we describe our configuration of each tested network in Appendix C.

5 Discussion

We now discuss the limitation of our work. As described in Section 3.2, our batch size parameter δ is bounded to 5 at maximum for both precision and time-efficiency concern. In consideration for precision solely, as we compute the over-approximate convex hull of the union of multiple convex polytopes, the process will inevitably introduce additional over-approximate error into the LP encoding, yielding coarser neuron intervals. Thus we bound the value of δ to mitigate the degree of precision sacrifice. For time-efficiency issue, the transformation between V-representation and H-representation (refer to Section 3.2) – in either direction – is generally NP-hard, thus incurring exponential overhead with larger dimensions. As the parameter δ yields a $(\delta + 1)$-dimensional space, it is advisable to keep δ-value small so that the convex hull computation process will not become an execution bottleneck. For future work, we will explore the possibility of assigning δ dynamically for different networks to strike a better trade-off between speed and precision.

Our proposed refinement process could be applied to other verification techniques for improved precision, as long as they use linear constraints to approximate the underlying network [6,8,9].

6 Related Work

Network verification methods can be generally categorized as complete or incomplete methods. Complete methods conduct exact analysis over the network, especially for ReLU-activated networks. Given adequate time and resources, the complete methods return deterministic verification or violation of the robustness property to the user. Typical existing works are usually SMT (satisfiability modulo theory) based, MILP (mixed integer liner program) based or branch and bound (BaB) based [3,4,22,23]. For instance, β-CROWN [23] is a GPU-based verifier which uses branch and bound method to enable exact reasoning over the ReLU activation function. Furthermore, β-CROWN could also perform as an incomplete verifier with early termination.

On the other hand, the incomplete methods choose to over-approximate the non-linearity of the network using abstract interpretation or bound propagation etc [6,7,24,25]. They are faced with precision loss due to the over-approximation of network behaviour. Consequently, the analysis result becomes inconclusive when the incomplete verifiers fail to verify the property. To rectify this deficiency, researchers have proposed various techniques like [8–10]. The work in [8] presents a new convex relaxation method that considers multiple ReLUs jointly in order to capture the correlation between ReLU neurons in the same layer. This idea has been further developed in PRIMA [9] which reduces the complexity of

ReLU convex abstraction computation via a novel convex hull approximation algorithm. In comparison, DeepSRGR [10] elects to refine the abstraction in an iterative manner, where it repeatedly uses the spurious regions to stabilize the ReLU neurons until the abstraction is precise enough to eliminate the adversarial label linked to that specified spurious region. In our work, we combine both these refinement methods [9,10] to break the precision barrier and also leverages the double-description method to retain efficiency as well.

7 Conclusion

We leverage the double description method in convex polytope area to compute the convex hull of the union of multiple polytopes, making it amenable for eliminating multiple adversarial labels simultaneously and boosting the analysis efficiency. Furthermore, we combine the convex relaxation technique with the iterative abstract refinement method to improve the precision in abstract interpretation based verification system. We implemented our prototypical analyzer ARENA to conduct both robustness verification and falsification. Experiment results show affirmatively that ARENA enhances abstract refinement techniques by attaining better verification precision compared to DeepSRGR, with reasonable execution time; it also competes favourably in comparison with PRIMA and α, β-CROWN. Finally, it is also capable of detecting adversarial examples.

We believe that our proposed method can positively boost the effectiveness of sound but incomplete analyses and be applied to other methods that use linear constraints to approximate the network for effective precision enhancement.

8 Acknowledgement

We are grateful to Julian Rüth saraedum and Komei Fukuda for their prompt and patient answer to our queries on cddlib. And we appreciate Mark Niklas Müller's assistance to our queries on PRIMA.

This research is supported by a Singapore Ministry of Education Academic Research Fund Tier 1 T1-251RES2103. The second author is supported by a Singapore National Research Foundation Grant R-252-000-B90-279 for the project Singapore Blockchain Innovation Programme.

References

1. Kui Ren, Tianhang Zheng, Zhan Qin, and Xue Liu. Adversarial attacks and defenses in deep learning. *Engineering*, 6(3):346–360, 2020.
2. Xiaoyong Yuan, Pan He, Qile Zhu, and Xiaolin Li. Adversarial examples: Attacks and defenses for deep learning. *IEEE Transactions on Neural Networks and Learning Systems*, 30(9):2805–2824, 2019.

3. Vincent Tjeng, Kai Y. Xiao, and Russ Tedrake. Evaluating robustness of neural networks with mixed integer programming. In *International Conference on Learning Representations (ICLR)*. OpenReview.net, 2019.
4. Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In *International Conference on Computer Aided Verification (CAV)*, pages 97–117. Springer, 2017.
5. Luca Pulina and Armando Tacchella. An abstraction-refinement approach to verification of artificial neural networks. In *International Conference on Computer Aided Verification (CAV)*, pages 243–257. Springer, 2010.
6. Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin T. Vechev. AI2: safety and robustness certification of neural networks with abstract interpretation. In *IEEE Symposium on Security and Privacy (SP)*, pages 3–18. IEEE Computer Society, 2018.
7. Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin T. Vechev. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages*, 3(POPL):41:1–41:30, 2019.
8. Gagandeep Singh, Rupanshu Ganvir, Markus Püschel, and Martin T. Vechev. Beyond the single neuron convex barrier for neural network certification. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 15072–15083, 2019.
9. Mark Niklas Müller, Gleb Makarchuk, Gagandeep Singh, Markus Püschel, and Martin T. Vechev. PRIMA: general and precise neural network certification via scalable convex hull approximations. *Proc. ACM Program. Lang.*, 6(POPL):1–33, 2022.
10. Pengfei Yang, Renjue Li, Jianlin Li, Cheng-Chao Huang, Jingyi Wang, Jun Sun, Bai Xue, and Lijun Zhang. Improving neural network verification through spurious region guided refinement. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part I*, volume 12651 of *Lecture Notes in Computer Science*, pages 389–408. Springer, 2021.
11. Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.
12. Komei Fukuda and Alain Prodon. Double description method revisited. In Michel Deza, Reinhardt Euler, and Yannis Manoussakis, editors, *Combinatorics and Computer Science, 8th Franco-Japanese and 4th Franco-Chinese Conference, Brest, France, July 3-5, 1995, Selected Papers*, volume 1120 of *Lecture Notes in Computer Science*, pages 91–111. Springer, 1995.
13. CMU. Alpha-Beta-CROWN: A Fast and Scalable Neural Network Verifier with Efficient Bound Propagation, 2022. https://github.com/huanzhang12/alpha-beta-CROWN. Retrieved on Aug 11th, 2022.
14. Peter McMullen. Convex polytopes, by branko grunbaum, second edition (first edition (1967) written with the cooperation of v. l. klee, m. perles and g. c. shephard. *Comb. Probab. Comput.*, 14(4):623–626, 2005.

15. ETH. ETH Robustness Analyzer for Neural Networks (ERAN), 2022. https://github.com/eth-sri/eran. Retrieved on Aug 11th, 2022.
16. 3rd International Verification of Neural Networks Competition (VNN-COMP'22), 2022. https://sites.google.com/view/vnn2022. Retrieved on Aug 11th, 2022.
17. Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.
18. Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research).
19. Nicholas Carlini and David A. Wagner. Towards evaluating the robustness of neural networks. In *IEEE Symposium on Security and Privacy (SP)*, pages 39–57, 2017.
20. Matthew Mirman, Timon Gehr, and Martin T. Vechev. Differentiable abstract interpretation for provably robust neural networks. In *International Conference on Machine Learning (ICML)*, pages 3575–3583, 2018.
21. Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.
22. Elena Botoeva, Panagiotis Kouvaros, Jan Kronqvist, Alessio Lomuscio, and Ruth Misener. Efficient verification of relu-based neural networks via dependency analysis. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 3291–3299. AAAI Press, 2020.
23. Shiqi Wang, Huan Zhang, Kaidi Xu, Xue Lin, Suman Jana, Cho-Jui Hsieh, and J. Zico Kolter. Beta-crown: Efficient bound propagation with per-neuron split constraints for complete and incomplete neural network verification. *CoRR*, abs/2103.06624, 2021.
24. Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. Efficient neural network robustness certification with general activation functions. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 4944–4953, 2018.
25. Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Formal security analysis of neural networks using symbolic intervals. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 1599–1614. USENIX Association, 2018.
26. Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. Efficient neural network robustness certification with general activation functions. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 4944–4953, 2018.

A Back-substitution based bound computation in DeepPoly

DeepPoly contains a back-substitution process in order to obtain more precise neuron bounds. As described in Section 2.1, DeepPoly generates two symbolic constraints for each neuron, where the constraints are defined over the preceding connected neurons. To compute a more precise concrete bound, DeepPoly recursively substitutes the symbolic constraints *backward* layer by layer until the constraints are expressed in terms of the input neurons. During this process, the constraints defined over neurons in that specific layer are generated and DeepPoly leverages those constraints to evaluate concrete bound values. The most precise bound among all these layers will be selected as the final concrete bound for the neurons.

For example, we consider the expression $y_1 - y_2$ in Section 2.1 as an auxiliary affine neuron $x_A = y_1 - y_2$. With the concrete intervals of y_1, y_2, we can estimate the concrete bounds of x_A to be:

$$-1.2 = l_6 - u_7 \leq x_A = y_1 - y_2 \leq u_6 - l_7 = 4.8 \tag{3}$$

We can back-substitute for x_A until x_A is defined over x_3, x_4, where we have:

$$-1.2 \leq x_3 - 0.5x_4 - 1 \leq x_A \leq x_3 \leq 4.8 \tag{4}$$

Lastly, we express x_A over input neurons x_1, x_2 and obtain:

$$-0.2 \leq 0.5x_1 + 1.5x_2 + 1.8 \leq x_A \leq x_1 + x_2 + 2.8 \leq 4.8 \tag{5}$$

As can be seen, the best lower bound we obtain is -0.2. Therefore, DeepPoly will return -0.2 for the lower bound of expression $y_1 - y_2$ in the example in Section 2.1. Similar process will be applied to compute the lower bound of expression $y_1 - y_3$.

B The algorithm of refinement process RefineWithKReLU

We hereby present the refinement function RefineWithKReLU in Algorithm 3, where it only differs from Algorithm 2 in the property encoding (line 3-4 in Algorithm 3 vs lines 3-4 in Algorithm 2). Algorithm 2 tries to encode multiple adversarial labels at the same time, while this method only encodes one adversarial label at one time (3-4).

In general, this method still aims to detect infeasibility of the network constraints and the negated property in order to refine the verification result (line 7-8). It leverages LP solving to refine the abstraction (line 9, 14), or find adversarial examples with optimization solution returned by LP solver (line 10, 15).

Algorithm 3: The algorithm for refinement process RefineWithKReLU

Function Name: RefineWithKReLU$(N, \mathit{\Pi}, L_c, \mathcal{L}_{adv}[i], \mathcal{L}_{elim})$

Input:

- N: input neural network with input layer γ_{in}, and output neurons $y_1, ..., y_n$
- $\mathit{\Pi}$: the constraint set of N
- y_c: the output neuron corresponding to the ground truth label L_c $(1 \leq c \leq n)$
- $\mathcal{L}_{adv}[i]$, $\mathcal{L}_{elim}$: the adversarial label to be refined, and the list of previously eliminated labels

Output: the refinement status and the number of iterations

```
1:  $counter = 0$
2:  while $counter < \tau$ do                                   // $\tau$ is an iteration threshold
3:      $y_j \leftarrow$ GetOutputNeuronOfLabel($\mathcal{L}_{adv}[i]$)
4:      $\mathit{\Sigma} \leftarrow \mathit{\Pi} \wedge (y_c - y_j \leq 0)$                        // Initialize constraint set
5:      for all $L_t \in \mathcal{L}_{elim}$ do
6:          $\mathit{\Sigma} = \mathit{\Sigma} \wedge (y_c - y_t > 0)$
7:      if IsInfeasible($\mathit{\Sigma}$) then
8:          return (Verified, $counter$)
9:      $N \leftarrow$ LPSolveInputInterval($\mathit{\Sigma}, \gamma_{in}$)      // Update network with new bounds
10:     if ExistsAnAdversarialExample($N$) then
11:         return (Falsified, $counter$)
12:     for all ReLU layer $\gamma'_k$ in $N$ do
13:         $\gamma_k \leftarrow$ GetPrecedingInputAffineLayer($\gamma'_k$)
14:         $N \leftarrow$ LPSolveUnstableReLUs($\mathit{\Sigma}, \gamma_k$)                 // Update new bounds
15:         if ExistsAnAdversarialExample($N$) then
16:             return (Falsified, $counter$)
17:     $A \leftarrow$ RecomputeNetworkAbstractionByDeepPoly($N$)
18:     $\mathit{\Pi} \leftarrow$ GetConstraintsInNetwork($N, A$)
19:     $counter = counter + 1$
20: return (Inconclusive, $counter$)
```

C Parameter setting for ARENA

Our tool ARENA contains the following parameters controlling over the analysis precision and efficiency:

- δ: defines the number of spurious regions to be considered simultaneously.
- cascade_flag: we mentioned two optimizations in Section 3.4. While optimization 1 is always activated, optimization 2 regarding cascading refinement can be deactivated by setting this flag to be false. By doing so, we directly enable the multiple adversarial label feature from the beginning of the refinement process to achieve better speed-up for larger networks.
- early_term_flag: we contain an early termination principle that if the number of updated ReLU neurons in the current iteration is zero, then we assume that this image is extremely hard to be verified and early terminate the analysis process without further time consumption; the principle will be enabled while the flag is set to be true.
- sparse_n: the number of neurons per layer to be grouped by 3-ReLU.

We demonstrate the parameter setting for each of the tested networks as follows in Table 7.

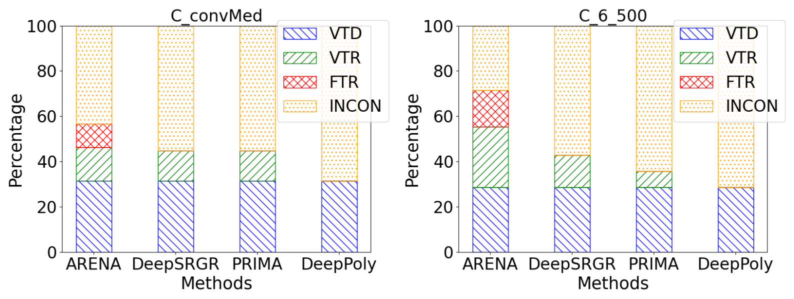
Neural Network	**batch_size** δ	**cascade_flag**	**early_term_flag**	**sparse_n**
M_3_100	4	true	false	70
M_5_100	5	true	false	70
M_6_100	5	true	false	70
M_9_100	5	false	true	70
M_6_200	5	false	true	70
M_9_200	5	false	true	70
M_convSmall	5	true	true	100
M_convMed	5	true	true	100
M_convBig	5	false	true	0
C_6_500	5	false	true	80
C_convMed	5	true	true	100

Table 7: The parameter configuration of ARENA

D Experiment result visualization

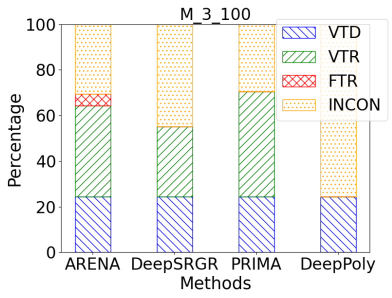
Our analysis results for MNIST networks are better visualized in Figure 10, where we report in percentage the number of testing images with different results. As ARENA, DeepSRGR and PRIMA all invoke DeepPoly first, the verified images by these three tools can be divided into these categories: the number of images which are **v**erified **t**hrough **D**eepPoly (abbreviated as VTD in Figure 10), **v**erified **t**hrough the **r**efinement process (VTR), **f**alsified **t**hrough the **r**efinement process (FTR) and **incon**clusive images (INCON).

Similarly, the visualized results of CIFAR10 networks are shown in Figure 9.

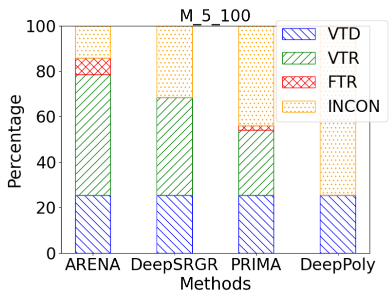


(a) (b)

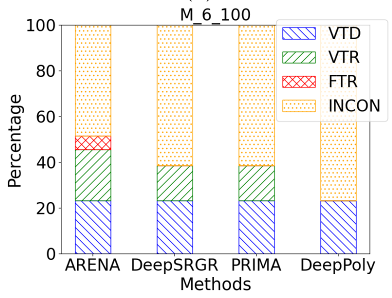
Fig. 9: Result comparison for CIFAR10 networks between ARENA, DeepSRGR, PRIMA and DeepPoly.



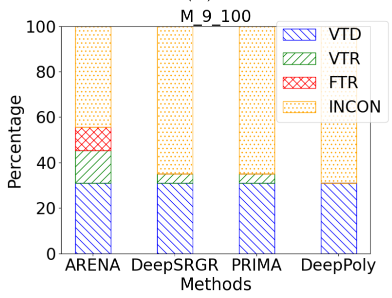
(a)



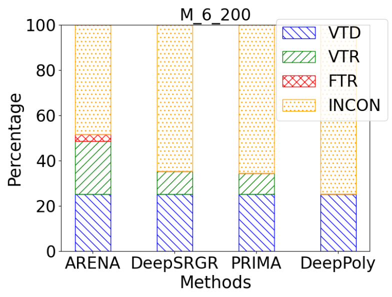
(b)



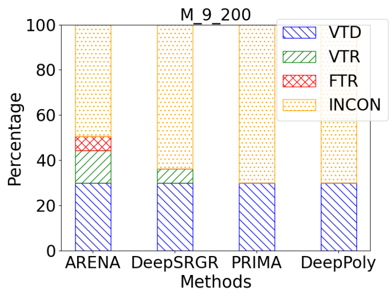
(c)



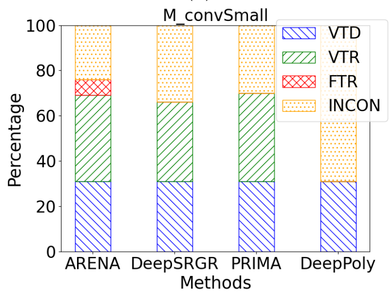
(d)



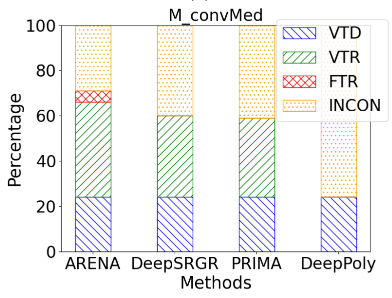
(e)



(f)



(g)



(h)

Fig. 10: Result comparison for MNIST networks between ARENA, DeepSRGR, PRIMA and DeepPoly.

We finally present the conclusive ratio of different tools for different networks in Table 8, where the conclusive ratio is computed as:

$$\frac{\text{the number of verified images} + \text{the number of falsified images}}{\text{the number of candidate images}} \tag{6}$$

Neural Network	ϵ	**ARENA**	**DeepSRGR**	**PRIMA**
M_3_100	0.028	69.4%	55.1%	70.4%
M_5_100	0.08	85.7%	68.3%	56.1%
M_6_100	0.025	51.5%	38.3%	38.3%
M_9_100	0.023	55.7%	35.1%	35.1%
M_6_200	0.016	51.5%	35.3%	34.3%
M_9_200	0.015	50.5%	36.1%	29.9%
M_convSmall	0.11	76%	66%	70%
M_convMed	0.1	71%	60%	59%
C_6_500	0.0032	71.4%	42.8%	35.7%
C_convMed	0.006	56.7%	44.7%	44.7%

Table 8: The conclusive ratio of ARENA, DeepSRGR and PRIMA

E Parameter setting and experiment results for PRIMA

The verification result of PRIMA depends on many parameters, we enumerate the important parameters as follows:

- k: the number of neurons in the k-ReLU group;
- sparse_n: the number of neurons per layer to be grouped by k-ReLU;
- partia_milp: the number of layers to be encoded using MILP (mixed integer liner program);
- max_milp_neurons: the max number of neurons to use for partial MILP encoding;
- timeout_final_lp: timeout for the final LP solver;
- timeout_final_milp: timeout for the final MILP solver.

In conducting our experiment, we deployed the parameter configuration for one of the convolutional networks reported in their paper [9]. Here, the parameters were set as k=3, sparse_n=100, partial_milp = 2, max_milp_neurons = 100, timeout_final_lp = 20, timeout_final_milp = 200, where the results are displayed in Table 9. In Section 4.2, we omitted the "Falsified" column for PRIMA since only 2 unsafe image is found for one MNIST network.

Neural Net	ϵ	**PRIMA**		
		Verified	Falsified	Time(s)
M_3_100	0.028	69	0	123.73
M_5_100	0.08	53	2	19.15
M_6_100	0.025	38	0	173.03
M_9_100	0.023	34	0	191.60
M_6_200	0.016	34	0	222.45
M_9_200	0.015	29	0	238.63
M_convSmall	0.11	70	0	84.23
M_convMed	0.1	59	0	125.88
C_6_500	0.0032	20	0	269.96
C_convMed	0.006	30	0	230.74

Table 9: The execution results for PRIMA

F Parameter setting for α,β-CROWN

α,β-CROWN (alpha-beta-CROWN) is a neural network verifier based on an efficient bound propagation algorithm [26] and branch and bound. It can be accelerated efficiently on GPUs and can scale to relatively large convolutional networks [13]. We selected five MNIST networks from ERAN benchmark [15] to evaluate the performance of α,β-CROWN: M_3_100, M_5_100, M_convSmall, M_convMed and M_convBig.

For M_3_100 and M_5_100, we used the experiment configuration of a network with similar size provided in their website [8], and only changed their corresponding epsilon and timeout. For M_convSmall and M_convMed, we deployed the designated configuration of M_convSmall [9], and changed their corresponding epsilon and timeout. For M_convBig, we also deployed the designated configuration of itself [10] and only changed the epsilon.

The experiment configurations of each network are displayed as follows:

```
# configuration of M_3_100
general:
  mode: verified-acc
  complete_verifier: bab-refine
model:
  name: mnist_3_100
  path: models/eran/mnist_relu_3_100.onnx
data:
```

[8] https://github.com/huanzhang12/alpha-beta-CROWN/blob/main/complete_verifier/exp_configs/mnist_6_100.yaml

[9] https://github.com/huanzhang12/alpha-beta-CROWN/blob/main/complete_verifier/exp_configs/mnist_conv_small.yaml

[10] https://github.com/huanzhang12/alpha-beta-CROWN/blob/main/complete_verifier/exp_configs/mnist_conv_big.yaml

```
  dataset: MNIST_ERAN_UN
  std: [1.0]
  mean: [0.0]
specification:
  epsilon: 0.028
attack:
  pgd_order: after
solver:
  beta-crown:
    batch_size: 1024
    iteration: 20
  mip:
    parallel_solvers: 16
    refine_neuron_time_percentage: 0.8
bab:
  timeout: 400
  branching:
    reduceop: max

# configuration of M_5_100
general:
  mode: verified-acc
  complete_verifier: bab-refine
model:
  name: mnist_5_100
  path: models/eran/mnist_relu_5_100.onnx
data:
  dataset: MNIST_ERAN_UN
  std: [1.0]
  mean: [0.0]
specification:
  epsilon: 0.08
attack:
  pgd_order: after
solver:
  beta-crown:
    batch_size: 1024
    iteration: 20
  mip:
    parallel_solvers: 16
    refine_neuron_time_percentage: 0.8
bab:
  timeout: 800
  branching:
    reduceop: max

# configuration of M_convSmall
general:
  mode: verified-acc
model:
  name: mnist_conv_small
  path: models/eran/mnist_convSmallRELU__Point.onnx
data:
  dataset: MNIST_ERAN
  std: [0.30810001492500305]
  mean: [0.1307000070810318]
specification:
  epsilon: 0.11
attack:
  pgd_restarts: 100
solver:
  beta-crown:
    batch_size: 2048
    iteration: 20
bab:
  timeout: 2000
  branching:
    reduceop: max
```

```
# configuration of M_convMed
general:
  mode: verified-acc
model:
  name: mnist_conv_med
  path: models/eran/mnist_convMedGRELU__Point.onnx
data:
  dataset: MNIST_ERAN
  std: [0.30810001492500305]
  mean: [0.1307000070810318]
specification:
  epsilon: 0.1
attack:
  pgd_restarts: 100
solver:
  beta-crown:
    batch_size: 2048
    iteration: 20
bab:
  timeout: 2000
  branching:
    reduceop: max

# configuration of M_convBig
general:
  mode: verified-acc
model:
  name: mnist_conv_big
  path: models/eran/mnist_conv_big_diffai.pth
data:
  dataset: MNIST_ERAN
specification:
  epsilon: 0.306
attack:
  pgd_restarts: 100
solver:
  beta-crown:
    batch_size: 1024
    iteration: 20
bab:
  timeout: 180
  branching:
    reduceop: max
```