# Improved Geometric Path Enumeration for Verifying ReLU Neural Networks


Stanley Bak[1], Hoang-Dung Tran[2,3],
Kerianne Hobbs[4,5], and Taylor T. Johnson[3]

[1] Stony Brook University
[2] University of Nebraska - Lincoln
[3] Vanderbilt Univeristy
[4] Air Force Research Laboratory
[5] Georgia Institute of Technology

**Abstract.** Neural networks provide quick approximations to complex functions, and have been increasingly used in perception as well as control tasks. For use in mission-critical and safety-critical applications, however, it is important to be able to analyze what a neural network can and cannot do. For feed-forward neural networks with ReLU activation functions, although exact analysis is NP-complete, recently-proposed verification methods can sometimes succeed.

The main practical problem with neural network verification is excessive analysis runtime. Even on small networks, tools that are theoretically complete can sometimes run for days without producing a result. In this paper, we work to address the runtime problem by improving upon a recently-proposed geometric path enumeration method. Through a series of optimizations, several of which are new algorithmic improvements, we demonstrate significant speed improvement of exact analysis on the well-studied ACAS Xu benchmarks, sometimes hundreds of times faster than the original implementation. On more difficult benchmark instances, our optimized approach is often the fastest, even outperforming inexact methods that leverage overapproximation and refinement.

## 1 Introduction

Neural networks have surged in popularity due to their ability to learn complex function approximations from data. This ability has led to their proposed application in perception and control decision systems, which are sometimes safety-critical. For use in safety-critical applications, it is important to prove properties about neural networks rather than treating them as black-box components.

A recent method [24] based on path enumeration and geometric set propagation has shown that exact analysis can be practical for piecewise linear neural networks. This includes networks with fully-connected layers, convolutional

layers, average and max pooling layers, and neurons with ReLU activation functions. Here, we focus on fully-connected layers with ReLU activation functions. The verification problem in this method is presented in terms of input / output properties of the neural network. The method works by taking the input set of states and performing a set-based execution of the neural network. Due to the linear nature of the set representation and the piecewise linear nature of the ReLU activation function, the set may need to be split after each neuron is executed, so that the output after the final layer is a collection of sets that can each be checked for intersection with an unsafe set.

Since the formal verification problem we are addressing has been shown to be NP-Complete [13], we instead focus on improving practical scalability. This requires us to choose a set of benchmarks for evaluation. For this, we focus on properties from the well-studied ACAS Xu system [13]. This contains a mix of safe and unsafe instances, where the original verification times measured from seconds to days, including some unsolved instances.

The main contributions of this paper are:

- several new speed improvements to the path enumeration method, along with correctness justifications, that are each systematically evaluated;
- the first verification method that verifies all 180 benchmark instances from ACAS Xu properties 1-4, each in under 10 minutes on a standard laptop;
- a comparison with other recent tools, including Marabou, Neurify, NNV, and ERAN, where our method is often the fastest and over 100x faster than the original path enumeration method implementation in NNV.

This paper first reviews background related to neural networks, the path enumeration verification approach, and the ACAS Xu benchmarks in Section 2. Next, Section 3 analyzes several algorithmic optimizations to the basic procedure, and systematically evaluates each optimization's effect on the execution times of the ACAS Xu benchmarks. A comparison with other tools is provided in Section 4, followed by review of related work in Section 5 and a conclusion.

## 2 Background

We now review the neural network verification problem (Section 2.1), the basic geometric path enumeration algorithm (Section 2.2), important spatial data structures (Section 2.3), and the ACAS Xu benchmarks (Section 2.4).

### 2.1 Neural Networks and Verification

In this work, we focus our attention on fully-connected, feedforward neural networks with ReLU activation functions. A neural network computes a function $\mathsf{NN} : \mathbb{R}^{n_i} \rightarrow \mathbb{R}^{n_o}$, where $n_i$ is the number of inputs and $n_o$ is the number of outputs. A neural network consists of $k$ layers, where each layer $i$ is defined with a weight matrix $W_i$ and a bias vector $b_i$. Given an input point $y_0 \in \mathbb{R}^{n_i}$, a neural network will compute an output point $y_k \in \mathbb{R}^{n_o}$ as follows:

$$\begin{aligned} x^{(1)} &= W_1 y_0 + b_1, \quad & y_1 &= f(x^{(1)}) \\ x^{(2)} &= W_2 y_1 + b_2, \quad & y_2 &= f(x^{(2)}) \\ &\vdots & & \\ x^{(k)} &= W_k y_{k-1} + b_k, \quad & y_k &= f(x^{(k)}) \end{aligned}$$

We call $y_{i-1}$ and $y_i$ the input and output of the $i$-th layer, respectively, and $x^{(i)}$ the intermediate values at layer $i$. The vector-function $f$ is defined using a so-called *activation function*, that is applied element-wise to the vector of intermediate values at each layer. We focus on the popular rectified linear unit (ReLU) activation function, $\mathsf{ReLU}(x) = \max(x, 0)$.

For this computation definition to make sense, the sizes of the weights matrices and bias vectors are restricted. The first layer must accept $n_i$-dimensional inputs, the final layer must produce $n_o$-dimensional outputs, and the intermediate layers must have weights and biases that have sizes compatible with their immediate neighbors, in the sense of matrix/vector multiplication and addition. The number of neurons (sometimes called hidden units) at layer $i$ is defined as the number of elements in the layer's output vector $y_i$.

**Definition 1 (Output Range).** *Given a neural network that computes the function* $\mathsf{NN}$ *and an input set* $\mathcal{I} \subseteq \mathbb{R}^{n_i}$*, the* ***output range*** *is the set of possible outputs of the network, when executed from a point inside the input set,* $\mathsf{Range}(\mathsf{NN}, \mathcal{I}) = \{y_k \mid y_k = \mathsf{NN}(y_0),\ y_0 \in \mathcal{I}\}$.

Computing the output range is one way to solve the verification problem.

**Definition 2 (Verification Problem for Neural Networks).** *Given a neural network that computes the function* $\mathsf{NN}$*, an input set* $\mathcal{I} \subseteq \mathbb{R}^{n_i}$*, and an unsafe set* $\mathcal{U} \subseteq \mathbb{R}^{n_o}$*, the* ***verification problem for neural networks*** *is to check if* $\mathsf{Range}(\mathsf{NN}, \mathcal{I}) \cap \mathcal{U} = \emptyset$.

If verification is impossible, we would also prefer to generate a counterexample $y_0 \in \mathcal{I}$ where $y_k = \mathsf{NN}(y_0)$ and $y_k \in \mathcal{U}$, although not all tools do this. We also further assume in this work that the input and unsafe sets are defined with linear constraints, $\mathcal{I} = \{x \mid A_i x \leq b_i, x \in \mathbb{R}^{n_i}\}$, and $\mathcal{U} = \{x \mid A_u x \leq b_u, x \in \mathbb{R}^{n_o}\}$.

### 2.2 Basic Geometric Path Enumeration Algorithm

Given enough time, the output range of a neural network can be computed exactly using a recently-proposed geometric path enumeration approach [24]. The general strategy is to execute the neural network with *sets* instead of points. A *spatial data structure* is used to represent the input set of states, and this set is propagated through each layer of the neural network, computing the set of

```
   input  : Input Set: 𝓘, Unsafe Set: 𝓤
   output: Verification Result (safe or unsafe)
1  s ← ⟨layer:0, neuron:None, θ : convert(𝓘)⟩ // computation-state tuple
2  𝓦 ← List() // initialize waiting list
3  𝓦.put(s)
4  result ← safe
5  while result = safe and ¬𝓦.empty() do
6  |  s ← 𝓦.pop()
7  |  result ← step(s,𝓦,𝓤) // updates 𝓦, given in Algorithm 2
8  end
9  return result
```

**Algorithm 1:** High-level neural-network path enumeration algorithm.

possible intermediate values and then the set of possible outputs repeatedly until the output of the final layer is computed. In this context, a spatial data structure represents some subset of states in a Euclidean space $\mathbb{R}^n$, where the number of dimensions $n$ is the number of neurons in one of the layers of the network, and may change as the set is propagated layer by layer. An example spatial data structure could be a polytope defined using a finite set of half-spaces (linear constraints), although as explained later this is not the most efficient choice. Section 2.3 will discuss spatial data structures in more detail.

The high-level verification method is shown in Algorithm 1, where functions in red are custom to the spatial data structure being used. The `convert` function (line 1) converts the input set $\mathcal{I}$ from linear constraints to the desired spatial data structure, and stores it in the $\theta$ element of $s$, where $s$ is called a *computation-state tuple*. A `neuron` value of `None` in the tuple indicates that next operation should be an affine transformation. The computation-state tuple is then put into a waiting list (line 3), which stores tuples that need further processing. The `step` function (line 7) propagates the set $\theta$ by a single neuron in a single layer of the network, and is elaborated on in the next paragraph. This function can modify $\mathcal{W}$, possibly inserting one or more computation-state tuples, although always at a point further along in the network (with a larger layer number or neuron index), which ensures eventual termination of the loop. This function will also check if the set, after being fully propagated through the network, intersects the unsafe set. In this case, `step` will return `unsafe`, which causes the `while` loop to immediately terminate since the result is known.

The `step` function propagates the set of states $\theta$ by one neuron, and is shown in Algorithm 2. The intermediate values are computed from the input set of each layer by calling `affine_transformation` (line 12). For the current neuron index $n$, the algorithm will check if the input to the ReLU activation function, dimension $n$ of the set $\theta$, is always positive (or zero), always negative, or can be either positive or negative. This is done by the `get_sign` function (line 21), which returns `pos`, `neg`, or `posneg`, respectively. In the first two cases, the current dimension $n$ of the set is left alone or assigned to zero (using the `project_to_zero` method), to reflect the semantics of the ReLU activation function when the input is positive or negative, respectively. In the third case, the set is split into

```
   input : Computation-State Tuple: s, Waiting List: 𝒲, Unsafe Set: 𝒰
   output: Safe so far? (safe or unsafe)
 1 if s.neuron = None then
 2 |  // finished with the previous layer
 3 |  if s.layer = k then
 4 |  |  // finished with all layers
 5 |  |  if s.θ.has_intersection(𝒰) = ∅ then
 6 |  |  |  return safe
 7 |  |  else
 8 |  |  |  return unsafe // alternatively, return counterexample here
 9 |  |  end
10 |  else
11 |  |  s.layer ← s.layer + 1
12 |  |  s.θ.affine_transformation(W_{s.layer}, b_{s.layer})
13 |  |  s.neuron ← 1
14 |  end
15 end
16 n ← s.neuron
17 s.neuron ← n + 1
18 if s.neuron > size(b_{s.layer}) then
19 |  s.neuron ← None // n is the last neuron in the current layer
20 end
21 switch get_sign(s, n) do
22 |  case pos do
23 |  |  // do nothing
24 |  case neg do
25 |  |  s.θ.project_to_zero(n)
26 |  case posneg do
27 |  |  t ← ⟨s.layer, s.neuron, s.θ⟩ // deep copy s
28 |  |  s.θ.add_constraint(n, ≥, 0) // split on positive case
29 |  |  t.θ.add_constraint(n, ≤, 0) // split on negative case
30 |  |  t.θ.project_to_zero(n)
31 |  |  𝒲.put(t)
32 end
33 𝒲.put(s)
34 return safe // safe so far
```

**Algorithm 2:** Pseudocode for `step` function, which propagates a set through the network by one neuron.

two sets along linear constraint where the input to the activation function equals zero. In the case where the input to the activation function is less than zero, the value of dimension $n$ is projected to zero, reflecting the semantics of the ReLU activation function. The splitting is done using the `add_constraint` method of the spatial data structure, which takes three arguments: $n$, `sign`, and `val`. This method intersects the set with the linear condition that the $n$-th dimension is, depending on `sign`, greater than, less than, and/or equal to `val`. Once the set has been propagated through the whole network, it is checked for intersection with the unsafe set (line 5), using the `has_intersection` method.

This enumeration algorithm has been shown to be sound and complete [24]. However, for this strategy to work in practice, the spatial data structure used to store $\theta$ must support certain operations *efficiently*. These are denoted in red in Algorithms 1 and 2: `convert`, `has_intersection`, `affine_transformation`, `get_sign`, `project_to_zero`, and `add_constraint`. Polytopes represented with half-spaces, for example, do not have a known efficient way to compute general affine transformations in high dimensions. Instead, linear star sets [4] will be used, which are a spatial data structure that support all the required operations efficiently and without overapproximation error. These will be elaborated on more in the next subsection.

In this work, we focus on optimizations to the presented algorithm that increase its practical scalability, while exploring the same set of paths. The most important factor that we do not control and influences whether this can succeed is the number of paths that exist. Each output set that gets checked for intersection with the unsafe set corresponds to a unique *path* through the network, where the path is defined by the sign of each element of the intermediate values vector at each layer. The algorithm enumerates every path of the network for a given input set. An upper bound on this is $2^N$, where $N$ is the total number of neurons in all the layers of the network. For many practical verification problem instances, however, the actual number of unique paths is significantly smaller than the upper bound.

### 2.3 Spatial Data Structures

Using the correct spatial data structure (set representation in this context) is important to the efficiency of Algorithm 1 and 2, as well as some of our optimizations. Here we review two important spatial data structures, zonotopes and (linear) star sets.

**Zonotopes.** A *zonotope* is an affine transformation of the $[-1, 1]^p$ box. Zonotopes have been used for efficient analysis of hybrid systems [8] as well as more recently to verify neural networks using overapproximations [7,21]. Zonotopes can be described mathematically as $Z = (c, G)$, where the *center* $c$ is an $n$-dimensional vector and *generator matrix* $G$ is an $n \times p$ matrix. The columns of $G$ are sometimes referred to as *generators* of the zonotope, and we write these as $g_1, \ldots, g_p$. A zonotope $Z$ encodes a set of states as:

$$Z = \left\{ x \in \mathbb{R}^n \;\middle|\; x = c + G\alpha,\ \alpha \in [-1, 1]^p \right\} \tag{1}$$

The two most important properties of zonotopes for the purposes of verification are that they are efficient for (i) affine transformation, and (ii) optimization.

An affine transformation of an $n$-dimensional point $x$ to a $q$-dimensional space is defined with a $q \times n$ matrix $A$ and $q$-dimensional vector $b$ so that the transformed point is $x' = Ax+b$. An affine transformation of every point in an $n$-dimensional set of points described by a zonotope $Z = (c, G)$ is easily computed as $Z' = (Ac + b, AG)$. Note this uses standard matrix operations which scale polynomially with the dimension of $A$, and are especially efficient if the number

of generators is small. In the verification problem, the number of generators, $p$, corresponds to the degrees of freedom needed to encode the input set of states. In ACAS Xu system, for example, there are 5 inputs, and so the input set can be encoded with 5 generators. In contrast, affine transformations of polytopes require converting between a half-space and vertex representation, which is slow.

The second efficient operation for zonotopes is optimization in some direction vector $v$. Given a zonotope $Z = (c, G)$ and a direction $v$ to maximize, the point $x^* \in Z$ that maximizes the dot product $v \cdot x^*$ can be obtained as a simple summation $x^* = c + \sum_{i=1}^{p} x_i^*$, where each $x_i^*$ is given as:

$$x_i^* = \begin{cases} v_i, & \text{if } v_i \cdot g_i \geq -v_i \cdot g_i \\ -v_i, & \text{otherwise} \end{cases} \tag{2}$$

**Star Sets.** A (linear) *star set* is another spatial data structure that generalizes a zonotope. A star set is an affine transformation of an arbitrary $p$-dimensional polytope. Mathematically, a star set $S$ is a 3-tuple, $(c, G, P)$, where $c$ and $G$ are the same as with a zonotope, and $P$ is a half-space polytope in $p$ dimensions. A star set $S$ encodes a set of states (compare with Equation 1):

$$S = \{x \in \mathbb{R}^n \mid x = c + G\alpha,\ \alpha \in P\} \tag{3}$$

A star set can encode any zonotope by letting $P$ be the $[-1, 1]^p$ box. Star sets can also encode more general sets than zonotopes by using a more complex polytope $P$. A triangle, for example, can be encoded as a star set by setting $P$ to be a triangle, using the origin as $c$ and the identity matrix as $V$. This cannot be encoded with zonotopes, as they must be centrally symmetric. In Algorithm 1 on line 1, the `convert` function produces the input star set $(c, G, P)$ from input polytope $\mathcal{I}$ setting $c$ to the zero vector, $G$ to the identity matrix, and $P$ to $\mathcal{I}$.

Affine transformations by a $q \times n$ matrix $A$ and $q$-dimensional vector $b$ of a star set $S$ can be computed efficiently similar to a zonotope: $S' = (Ac+b, AG, P)$.

Optimization in some direction $v$ is slightly less efficient than with a zonotope, and can be done using linear programming (LP). To find a point $x^* \in S$ that maximizes the dot product $v \cdot x^*$, we convert the optimization direction $v$ to the initial space $w = (vG)^T$, find a point $\alpha^* \in P$ that maximizes $w$ using LP, and then convert $\alpha^*$ back to the $n$-dimensional space $x^* = c + G\alpha^*$.

Star sets, unlike zonotopes, also efficiently support half-space intersection operations by adding constraints to the star set's polytope. Given a star set $S = (c, G, P)$ and an $n$-dimensional half-space $dx \leq \mathtt{e}$ defined by vector $d$ and scalar $\mathtt{e}$, we convert this to a $p$-dimensional half-space as follows:

$$(dG)\alpha \leq \mathtt{e} - dc \tag{4}$$

The star set after intersection is then $S' = (c, G, P')$, where the half-space polytope $P'$ is the same as $P$, with one additional constraint given by Equation 4.

### 2.4 ACAS Xu Benchmarks

Since the verification problem for neural networks is NP-Complete, we know exact analysis methods cannot work well in all instances. In order to evaluate improvements, therefore, we must focus on a set of benchmarks.

In this work, we choose to focus on the Airborne Collision System X Unmanned (ACAS Xu) set of neural network verification benchmarks [13]. As these benchmarks have been widely-used for evaluation in other publications, and some authors have even made their tools available publicly, using these allows us to provide a common comparison point with other methods later in Section 4.

ACAS Xu is a flight-tested aircraft system designed to avoid midair collisions of unmanned aircraft by issuing horizontal maneuver advisories [17]. The system was designed using a partially observable Markov decision process that resulted in a 2GB lookup table which mapped states to commands. This mapping was compressed to 3MB using 45 neural networks (two of the inputs were discretized and are used to choose the applicable network) [12]. Since the compression is not exact, the verification step checks if the system still functions correctly.

Each network contains five inputs that get set to the current the aircraft state, and five outputs that determine the current advisory. The network has six ReLU layers with 50 neurons each, for a total of 300 neurons. Ten properties were originally defined, encoding things like, if the aircraft are approaching each other head-on, a turn command will be advised (property 3). The formal definition of all the properties encoded as linear constraints is available in the appendix of the original work [13].

## 3 Improvements

We now systematically explore several improvements to the exact path enumeration verification method from Section 2.2. For each proposed improvement, we compare the run-time on the ACAS Xu system with and without the change. We focus on properties 1-4. Although originally these were measured on a subset of the 45 networks [13], the same authors later used all the networks to check these properties [14], which is what we will do here. Each verification instance is run with a 10 minute timeout, so that the maximum time needed to test a single method, if a timeout is encountered on each of the 180 benchmarks, is 30 hours. Later, in Section 4, we will compare the most optimized method with other verification tools and the other ACAS Xu properties. Unless indicated otherwise, our experiments were performed on a `Laptop` platform with Ubuntu Linux 18.04, 32 GB RAM and an Intel Xeon E-2176M CPU running at 2.7 GHz with 6 physical cores (12 virtual cores with hyperthreading). The full data measurements summarized in this section are provided in Appendix C.

### 3.1 Local Search Type (DFS vs BFS)

Algorithm 1 uses a waiting list to store the computation-state tuples, which are popped off one at a time and passed to the `step` function. This need not strictly

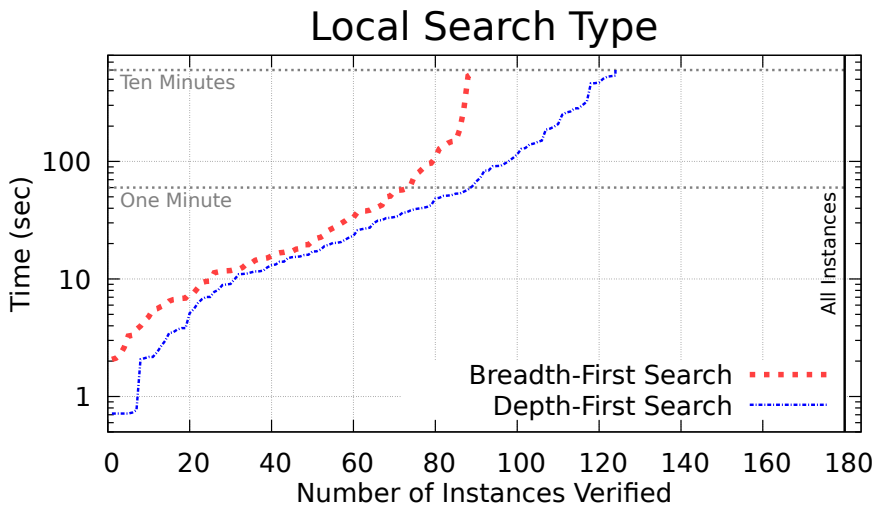


Fig. 1: Depth-first search outperforms breadth-first search.

be a list, but is rather a collection of computation-state tuples, and we can consider changing the order states are popped to explore the state space with different strategies. If the possible paths through the neural network are viewed as a tree, two well-known strategies for tree traversal that can be considered are depth-first search (DFS) and breadth-first search (BFS). A DFS search can be performed popping the computation-state tuple with the largest $(\mathsf{layer}, \mathsf{neuron})$ pair, whereas a BFS search is done by popping the tuple with the smallest $(\mathsf{layer}, \mathsf{neuron})$ pair.

The original path enumeration with star set approach [24] describes a layer-by-layer exploration strategy, which is closer to a BFS search. Finite-state machine model-checking methods, however, more often use DFS search.

We compare the two approaches in Figure 1, which summarizes the execution of all 180 benchmarks. Here, the $y$-axis is a timeout in seconds, and the $x$-axis is the number of benchmarks verified within that time. Within the ten minute timeout, around 90 benchmarks can be successfully verified with BFS, and 120 with DFS [6]. Notice that the $y$-axis is log scale, so that differences in runtimes between easy and hard benchmark instances are both visible.

As can be seen in the figure, the DFS strategy is superior. This is primarily due to unsafe instances of the benchmarks, where DFS can often quickly find an unsafe execution and exit the high-level loop, whereas BFS first iterates through all the layers and neurons (DFS explores deep paths, which sometimes are quickly found to be unsafe). In the cases where the system was safe, both approaches took similar time. Another known advantage of DFS search is that the memory

[6] The DFS method solves every benchmark that can be solved with BFS. Appendix C contains the complete results.

needed to store the waiting list is significantly smaller, which can be a factor for the benchmarks with a large number of paths.

**Correctness Justification:** Both DFS and BFS explore the same sets of states, just in a different order.

### 3.2 Bounds for Splitting

Using DFS search, we consider other improvements. The original path enumeration publication mentions the following optimization:

> "... to minimize the number of [operations] and computation time, we first determine the ranges of all states in the input set which can be done efficiently by solving ... linear programming problems." [24]

An evaluation of the improvement is not provided, so we investigate this here. The optimization is referring to the implementation of the `get_sign` function on line 21 of Algorithm 2. The `get_sign`$(s, n)$ function takes as input a computation-state tuple $s$ with spatial data structure $\theta$ (a star set) and a dimension number $n$. It returns pos, neg, or posneg, depending on whether value of dimension $n$, which we call $x_n$, in set $\theta$ can be positive (or zero), negative or both. Our baseline implementation, which we refer to as Copy, determines the output of `get_sign` by creating two copies of the passed-in star set, intersecting them with the condition that $x_n \leq 0$ or $x_n \geq 0$, and then checking each star set for feasibility, done using linear programming (LP). In the second version, which we call Bounds, the passed-in star set is instead minimized and maximized in the direction of $x_n$, to determine the possible signs. While Copy incurs overhead from creating copies and adding intersections, Bounds does extra work by computing the minimum and maximum which are not really needed (we only need the possible signs of $x_n$).

A comparison of the optimizations on the ACAS Xu benchmarks are shown in Figure 2. by comparing Copy to Bounds, we confirm the original paper's claim that Bounds is faster.

**Correctness Justification:** If $\theta$ intersected with $x_n \leq 0$ is feasible, then the minimum value of $x_n$ in $\theta$ will be less than or equal to zero and vice versa. Similar for the maximum case.

### 3.3 Fewer LPs with Concrete Simulations

We next consider strategies to determine the possible signs of a neuron's output with fewer LP calls, which we call *prefiltering*. Consider a modification of the Bounds optimization, where rather than computing both the upper and lower bound of $x_n$, we first compute the lower bound and check if its value is positive. If this is the case, we know `get_sign` should return pos, and we do not need to compute the upper bound. We could, alternatively, first compute the upper bound and check if its value is negative. If there is no branching and we guess the correct side to check, only a single LP needs to be solved instead of two.

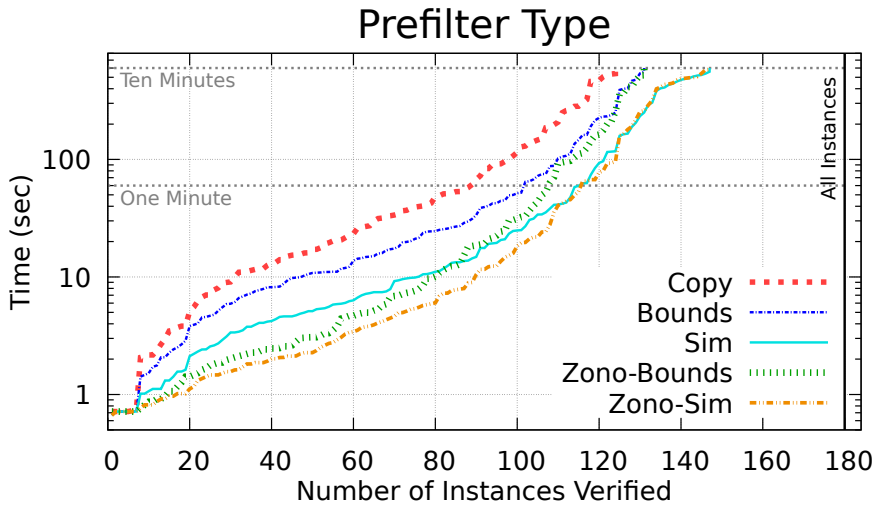


Fig. 2: Prefilter optimizations improve performance by rejecting branches without LP solving. The Zono-Sim method works best.

We can do even better than guessing by tracking extra information in the computation-state tuple. We add a `simulation` field to $s$, which contains a concrete value in the set of states $\theta$. This is initialized to any point in the input set $\mathcal{I}$, which can be obtained using LP, or using the center point if the input states are a box. When `get_sign` returns posneg and the set is split (line 27 in Algorithm 2), the optimization point $x^*$ that proved a split was possible is used as the value of `simulation` in the new set. Also, when an affine transformation of the set is computed (line 12 in Algorithm 2), or when the set is projected to zero, `simulation` must also be modified by the same transformation.

With a concrete value of $x_n$ available in `simulation`, we use its sign to decide whether to first check the upper or lower bound of dimension $n$ in $\theta$. If the $n$th element of `simulation` is positive, for example, we first compute the lower bound. If this is positive (or zero), then `get_sign` can return pos. If the lower bound is negative, then we can immediately return posneg without solving another LP, since the simulation serves as a witness that $x_n$ can also be positive. Only when the simulation value of $x_n$ is zero do we need to solve two LPs.

We call this method Sim in Figure 2. This is shown to be generally faster than the previous methods, as the overhead to track simulations is small compared with the gains of solving fewer LPs.

**Correctness Justification:** If the lower bound of $x_n$ is greater than zero, than its upper bound will be also be greater than zero and pos is the correct output. If the lower bound is less than zero and the $n$th element of `simulation` is greater than zero, than the upper bound will also be positive, since it must be greater than or equal to the value in the simulation (`simulation` is always a point in the set $\theta$), and so posneg is correct. Similar for the opposite case.

### 3.4 Zonotope Prefilter

We can further reduce LP solving by using a zonotope. In each computation-state tuple $s$, we add a zonotope field $z$ that overapproximates $\theta$, so that $\theta \subseteq z$. In the ACAS Xu benchmarks (and most current benchmarks for verification of NNs), the input set of states is provided as interval values on each input, which is a box and can be used to initialize the zonotope. Otherwise, LPs can be solved to compute box bounds on the input set to serve as an initial value. During the affine transformation of $\theta$ (line 12 in Algorithm 2), the zonotope also gets the same transformation applied. Cases where $\theta$ gets projected to zero are also affine transformations and can be exactly computed with the zonotope $z$. The only unsupported operation in the algorithm for zonotopes is `add_constraint`, used during the splitting operation (lines 28-29 in Algorithm 2). We skip these operations for the zonotope, which is why $z$ is an overapproximation of $\theta$.

With a zonotope overapproximation $z$ available during `get_sign`, we can sometimes reduce the number of LPs to zero. Computing the minimum and maximum of the $n$-th dimension of $z$ is an optimization problem over zonotopes, which recall from Section 2.3 can be done efficiently as a simple summation. If the $n$-th dimension of $z$ is completely positive or negative, we can return pos or neg immediately. Otherwise, if both positive and negative values are possible in the zonotope, we fall back to LP solving on $\theta$ to compute the possible signs. This can be done either by computing both bounds, which we call Zono-Bounds or with the simulation optimization from before, which we call Zono-Sim. The performance of the methods are shown in Figure 2. The Zonotope-Sim method performs the fastest, verifying about 145 benchmarks in under 10 minutes and demonstrating that reduction in LP solving is worth the extra bookkeeping.

**Correctness Justification:** Rejecting branches without LP solving is justified by the fact that $z$ is an overapproximation of $\theta$. This is initially true, as if the input set is a box then $z = \theta$ and otherwise $z$ is the box overapproximation of $\theta$. This is also true for every operation other than `add_constraint`, as these are exact for zonotopes. Finally, it is also true when `add_constraint` operation is skipped on $z$, as adding constraints can only reduce the size of the set $\theta$. If $\theta \subseteq z$, every smaller set $\theta'$ will also be a subset of $z$ by transitivity, $\theta' \subseteq \theta \subseteq z$, and so an overapproximation is maintained by ignoring these operations with $z$. Finally, if the $n$-th dimension of an overapproximation of $\theta$ is strictly positive (or negative), the $n$-th dimension of $\theta$ will also be strictly positive (or negative).

### 3.5 Eager Bounds Computation

The `step` function shown in Algorithm 2 computes the sign of $x_n$ for the current neuron $n$. An alternative approach is to compute the possible signs for every neuron's output in the current layer immediately after the affine transformation on line 12. These bounds can be saved in the computation-state tuple $s$ and then accessed by `get_sign`. The potential advantage is that, if a split is determined as impossible for some neuron $n$, and a split occurs at some earlier neuron $i < n$, then the split will also be impossible for neuron $n$ in both of the sets resulting

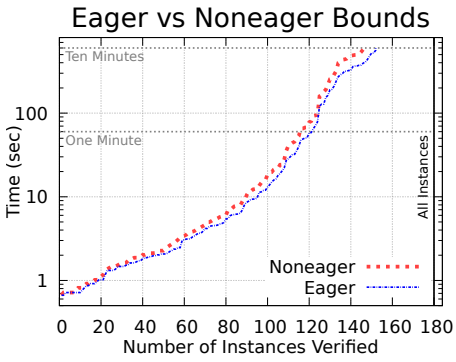


Fig. 3: Computing neuron output bounds eagerly improves speed.

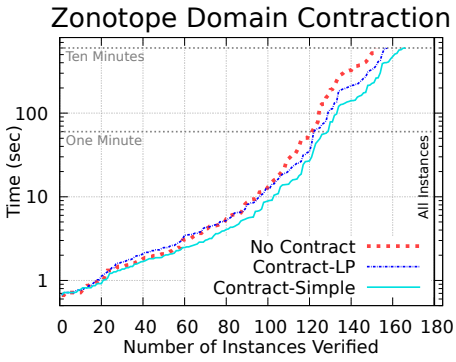


Fig. 4: Zonotope domain contraction improves overall performance.

from the earlier split at neuron $i$. In this way, computing the bounds once for neuron $n$ is sufficient in the parent set, as opposed to computing the bounds twice, in each of the two children sets resulting from the split. The benefit can be even more drastic if there are multiple splits before neuron $n$ is processed, where potentially an exponential number of bounds computations can be skipped due to a single computation in the parent. On the other hand, if a split is possible, we will have computed more bounds than we needed, as we will do the computation once in the parent and then once again in each of the children. Furthermore, this method incurs additional storage overhead for the bounds, as well as copy-time overhead when computation-state tuples are deep copied on line 27. Experiments are important to check if the benefits outweigh the costs.

The modified algorithm, which we call Eager, will use the zonotope prefilter and simulation as before to compute the bounds, but this will be done immediately after the affine transformation on line 12. Further, when a split occurs along neuron $n$ in the posneg case, the bounds also get recomputed in the two children for the remaining neurons in the layer, starting at the next neuron $n+1$. Neurons where a split was already rejected do not have their bounds recomputed. This algorithm is compared with the previous approach, called Noneager. In Figure 3, we see eager computation of bounds slightly improves performance.

**Correctness Justification:** When sets are split in the posneg case in Algorithm 2, each child's $\theta$ is a subset of the parent's $\theta$. Thus, the upper and lower bound of the output of each neuron $n$ can only move inward. Thus, if the parent's bounds for some neuron are strictly positive (or negative), then the two childrens' bounds will match the parent's and do not need to be recomputed.

### 3.6 Zonotope Contraction

The accuracy of the zonotope prefilters is important, as large overapproximation error will lead to the computed overapproximation range of $x_n$ in zonotope $z$ always overlapping zero, and thus performance similar to the Sim method. This effect is observed near the top of the curves in Figure 2.

In order to improve accuracy, we propose a zonotope domain contraction approach, where the size of the zonotope set $z$ is reduced while still maintaining an overapproximation of the exact star set $\theta$. As discussed before, computing exact intersections of zonotopes is generally impossible when splitting (lines 28-29 in Algorithm 2). However, we can lower our expectations and instead consider other ways to reduce the size of zonotope $z$ while maintaining $\theta \subseteq z$.

To do this, we use a slightly different definition of a zonotope, which we refer to as an *offset zonotope*. Instead of an affine transformation of the $[-1, 1]^p$ box, an offset zonotope is an affine transformation of an arbitrary box, $[l_1, u_1] \times \ldots \times [l_p, u_p]$, where each upper bound $u_i$ is greater than or equal to the lower bound $l_i$. As this corresponds to an affine transformation of the $[-1, 1]^p$ box, offset zonotopes are equally expressive as ordinary zonotopes. Optimization over offset zonotopes can also be done using a simple summation, but instead of using Equation 2, we use the following modified equation:

$$x_i^* = \begin{cases} u_i v_i, & \text{if } u_i v_i \cdot g_i \geq l_i v_i \cdot g_i \\ l_i v_i, & \text{otherwise} \end{cases} \tag{5}$$

Using offset zonotopes allows for some memory savings in the algorithm. The initial zonotope can be created using a zero vector as the zonotope center and the identity matrix as the generator matrix, the same as the initial input star set. In fact, with this approach, since the affine transformations being applied to the zonotope $z$ and star set $\theta$ are identical, the centers and generator matrices will always remain the same, so that we only need to store one copy of these.

Beyond memory savings, with offset zonotopes we can consider ways to reduce the zonotope's overapproximation error when adding constraints to $\theta$. The proposed computations are done after splitting (lines 28-29 in Algorithm 2), each time an extra constraint gets added to the star set's polytope $P$. The new linear constraint in the output space ($x_n \leq 0$ or $x_n \geq 0$) is transformed to a linear constraint in the initial space using Equation 4. We then try to contract the size of the zonotope's box domain by increasing each $l_i$ and reducing each $u_i$, while still maintaining an overapproximation of the intersection. We consider two ways to do this which we call Contract-LP and Contract-Simple.

In Contract-LP, linear programming is used to adjust each $l_i$ and $u_i$. Since the affine transformations for the star set $\theta$ and the zonotope $z$ are the same, $z$ is an overapproximation if and only if the star set's polytope $P$ is a subset of $z$'s initial domain box $[l_1, u_1] \times \ldots \times [l_p, u_p]$. Thus, we can compute tight box bounds on $P$ using linear programming, and using this box as the offset zonotope's initial domain box. This will be the smallest box that is possible for the current affine transformation while still maintaining an overapproximation. This approach, however, requires solving $2p$ linear programs, which may be expensive.

Another approach is possible without invoking LP, which we call Contract-Simple. Contract-Simple overapproximates the intersection by considering only the new linear constraint. This is a problem of finding the smallest box that

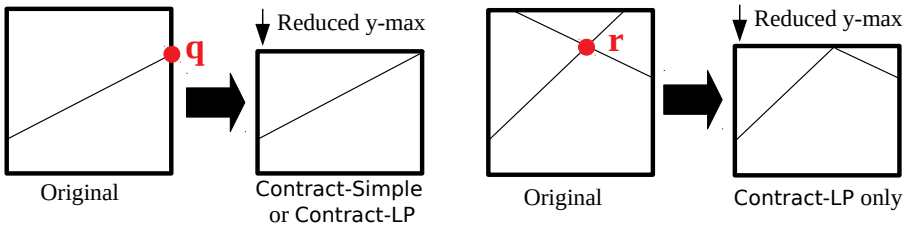


Fig. 5: Both Contract-Simple and Contract-LP can find point $q$ to contract a zonotope's initial box (left), but only Contract-LP can find point $r$ (right), as it requires reasoning with multiple linear constraints.

contains the intersection of an initial box and a single halfspace, which can be solved geometrically without LP solving (see Appendix A for an algorithm).

Since Contract-Simple only considers a single constraint, it can be less accurate than Contract-LP. An illustration of the two methods is given in Figure 5, where the initial domain is a two-dimensional box. The thin lines are the linear constraints that were added to $\theta$, where all points below these lines are in the corresponding halfspaces. On the left, both Contract-Simple and Contract-LP can reduce the upper bound in the $y$ direction by finding the point $q$, which lies at the intersection of one side of the original box domain and the new linear constraint. On the right, two constraints were added to the star $\theta$ (after two split operations), and they both must be considered at the same time to find point $r$ to be able to reduce the upper bound in the $y$ direction. In this case, only Contract-LP will succeed, as Contract-Simple works with only a single linear constraint at a time, and intersecting the original box with each of the constraints individually does not change its size.

Comparing the performance of the methods in Figure 4, we see that the less-accurate but faster Contract-Simple works best for the ACAS Xu benchmarks. We expect both methods to take longer when the input set has more dimensions, but especially Contract-LP since it requires solving two LPs for every dimension.

**Correctness Justification:** The domain contraction procedures reduces the size of zonotope $z$ while maintaining an overapproximation of the star set $\theta$. This can be seen since the affine transformations in $z$ and $\theta$ are always the same, and every point in the star set's initial input polytope $P$ is also a point in the initial box domain of $z$. Since an overapproximation of $\theta$ is maintained, it is still sound to use $z$ when determining the possible signs of a neuron's output.

## 4 Evaluation with Other Tools

We next compare the optimized implementation with other neural network verification tools. Our optimizations are part of the exact analysis mode of the NNENUM tool available at https://github.com/stanleybak/nnenum. The artifact evaluation package for our measurements here is online at http://stanleybak.com/papers/bak2020cav_repeatability.zip.

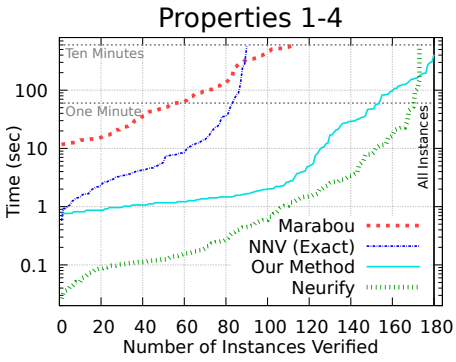


Fig. 6: Our method verifies all the benchmarks, although Neurify is usually faster when it completes.

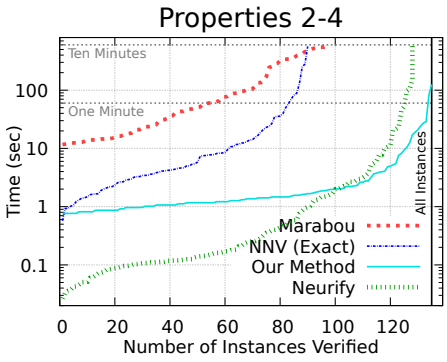


Fig. 7: Without property 1, our approach is generally fastest when the runtime exceeds two seconds.

We evaluate with the fully optimized method, using DFS local search, Zono-Sim prefilter, Eager bounds, Contract-Simple zonotope domain contraction. Further, we use a parallelized version of the algorithm, where the details of the parallalization are provided in Appendix B. With a 12-thread implementation (one for each core on our evaluation system), the algorithm can now verify all 180 ACAS Xu benchmarks from properties 1-4 within the 10 minute timeout. All measurements are done on our Laptop system, with hardware as described in the first paragraph of Section 3. The complete measurement data summarized here is available in Appendix D.

**ACAS Xu Properties 1-4.** We compare our method with Marabou [14] Neurify [26], and NNV [25]. Marabou is the newer, faster version of the Reluplex algorithm [13], where a Simplex-based LP solver is modified with special ReLU pivots[7]. Neurify is the newer, 20x faster version of the ReluVal algorithm [27], which does interval-based overapproximation, and splits intervals based on gradient information, ensuring the overapproximation error cannot cause to an incorrect result. NNV is the original Matlab implementation of the path enumeration method with star sets, available online at `https://github.com/verivital/nnv`. The verification result is consistent between the methods, which is a good sanity check for implementation correctness.

The comparison on ACAS Xu benchmarks on properties 1-4 is shown in Figure 6. Our method is the only approach able to analyze all 180 benchmarks in less than 10 minutes, and outperforms both Marabou and NNV.

The comparison with Neurify is more complicated. In Figure 6, Neurify was faster (when it finished) on all but the largest instances. One advantage of Neurify compared with the other tools is that if the unsafe set is very far away from the possible outputs of a neural network, it can prove safety quickly with a very coarse overapproximation. Path enumeration methods, on the other hand,

[7] For Marabou, we used the faster parallel divide-and-conquer mode with arguments as suggested in the paper [14]: `--dnc --initial-divides=4 --initial-timeout=5 --num-online-divides=4 --timeout-factor=1.5 --num-workers=12`.

Table 1: Tool runtime (secs) for ACAS Xu properties 5-10.

| Property | Net | Result | Our Method | ERAN | Neurify | NNV Exact | Marabou |
|---|---|---|---|---|---|---|---|
| 5 | 1-1 | SAFE | 13 | - | 12 | 671 | 1969 |
| 6.1 | 1-1 | SAFE | 67 | - | 3 | 6230 | 12425 |
| 6.2 | 1-1 | SAFE | 76 | - | 1 | 7612 | 17755 |
| 7 | 1-9 | UNSAFE | 5948 | - | 804 | - | - |
| 8 | 2-9 | UNSAFE | .7 | - | 64 | - | - |
| 9 | 3-3 | SAFE | 88 | 318 | 393 | 12576 | 15235 |
| 10 | 4-5 | SAFE | 12 | - | 1 | 457 | 2795 |

explore all paths regardless of the distance to the unsafe set. This is especially relevant for ACAS Xu property 1, where the system is unsafe if the first output, clear-of-conflict, is greater than 1500 whereas, for example on network 1-1, this output is always smaller than 1. The meaning of this property is also strange: the absolute value of a specific output is irrelevant, as relative values are used to select the current advisory. Neurify is admittedly the clear winner for all the networks with this property.

When this property is excluded and instead only the more difficult properties 2-4 are considered (Figure 7), a different trend emerges. Here, our method outperforms Neurify when analysis takes more than about two seconds, which we believe is an encouraging result. Further, part of the reason why Neurify can be very quick on the easier benchmarks (with runtime less than two seconds) is that our implementation incurs a startup delay of about 0.6 seconds simply to start the Python process and begin executing our script, by which time the C++-based Neurify can verify 80 benchmarks. We believe the more interesting cases are when the runtimes are large, and we outperform Neurify in these cases.

Finally, we compare with using single-set overapproximations for analysis. NNV provides an approximate-star method, where rather splitting, a single star set is used to overapproximate the result of ReLU operations. While fast when it succeeds, this strategy can only verify 68 of the 180 benchmarks. Furthermore, the benchmarks it verified were also quickly checked with exact path enumeration. Of the 68 verified benchmarks, the largest performance difference was property 3 with network 3-3, which took 3.1 seconds with exact enumeration and 1.2 seconds with single-set overapproximation. For these ACAS Xu benchmarks, overapproximation using a single set does not provide much benefit.

**Other ACAS Xu Properties.** Another recently proposed and well received analysis method is presented in the elegant framework of abstract interpretation using zonotopes, in tools such as AI$^2$ [7] or DeepZ [21]. These methods are single-set overapproximation methods, similar to the approximate-star method in NNV, but with strictly more error (see Figure 2 in the NNV paper [24] and the associated discussion). As these methods have more error than approximate-star, and since approximate-star could only verify 68 of the 180 benchmarks, we do not expect these methods to work well on the ACAS Xu system.

However, a recent extension to these methods has been proposed where the overapproximation is augmented with MILP solving [22] to provide complete analysis. This has been implemented in the ERAN tool, publicly available at https://github.com/eth-sri/eran. According to current version of the `README`, ERAN currently only supports property 9 of ACAS Xu, so we were unable to try this method on the other ACAS Xu networks or properties. Verifying property 9 uses a hard-coded custom strategy of first partitioning the input space into 6300 regions and analyzing these individually. This problem-specific parameter presents a problem for fair timing comparison, as the time needed to find the splitting parameter value of 6300 is unknown and does not get measured.

Ignoring this issue, we ran a comparison on property 9 and network 3-3, the only network where the property applies. A runtime comparison for ERAN[8] and the other tools is shown in Table 1. Surprisingly, our enumeration method significantly outperforms the overapproximation and refinement approaches both in Neurify and ERAN on this benchmark. Notice, however, that the original enumeration method in NNV is much slower than our method (about 150x slower in this case). Without the optimizations from this work, one would reach the opposite conclusion about which type of method works better for this benchmark. Both NNV and our method, however, report exploring the same number of paths, 338600 on this system.

For completeness, Table 1 also includes the other original ACAS Xu properties, which were each defined over a single network[9]. Both our method and Neurify completed all the benchmarks, although neither was best in all cases. Property 7 is particularly interesting, since the input set is the entire input space, so the number of path is very large. Hundreds of millions of paths were explored before finding a case where the property was violated.

## 5 Related Work

As the interest in neural networks has surged, so has research in their verification. We review some notable results here, although recent surveys may provide more a thorough overview [15,28]. Verification approaches for NNs can broadly be characterized into geometric techniques, SMT methods, and MILP approaches.

Geometric approaches, like this work, propagate sets of states layer by layer. This can be done with polytopes [29,6] using libraries like the multi-parametric toolbox (MPT) [10], although certain operations do not scale well, in particular, affine transformation. Other approaches use geometric methods to bound the range of a neural network. These include $AI^2$ [7] and DeepZ [21] which propagate zonotopes through networks and are presented in the framework of abstract interpretation. ReluVal [27] and Neurify [26] also fall into this category, using interval symbolic methods to create overapproximations, followed by a refinement strategy based on symbolic gradient information. Some of these

[8] For ACAS Xu analysis, we used the following arguments provided by the ERAN authors: `--domain deepzono --dataset acasxu --complete True`

[9] Property 6's input set was a disjunction of two boxes which we split into two cases.

implementations are also sound with respect to floating-point rounding errors, which we have not considered here, mostly for lack of an LP solver that is both fast and does outward rounding. Other NN verification tools such as Reluplex, Marabou, ERAN, and NNV also use numeric LP solving. Another performance difference is that we used the free GLPK library for LP solving and some other tools used the commercial Gurobi optimizer, which is likely faster. Other refinement approaches partition the input space to detect adversarial examples [11], compute maximum sensitivity for verification [30], or perform refinement based on optimization shadow prices [20].

Mixed integer-linear programming (MILP) solvers can be used to exactly encode the reachable set of states through a ReLU network using the big-M trick to encode the possible branches [16,23]. This introduces a new boolean variables for each neuron, which may limit scalability. The MILP approach has also been combined with a local search [5] that uses gradient information to speed up the search process.

SMT approaches include the Reluplex [13] and Marabou [14], which modify the Simplex linear programming algorithm by splitting nodes into two, which are linked by the semantics of a ReLU. The search process is modified with updates that fix the ReLU semantics for the node pairs. Another tool, Planet, combines the MILP approach with with SAT solving and linear overapproximation [6].

Here, we focused on input/output properties of the neural network, given as linear constraints. This formulation can check for adversarial examples [9] in image classification within some $L_\infty$ norm of a base image, which are essentially box input sets. Other more meaningful semantic image perturbations such as rotations, color shifting, and lighting adjustments can also be converted into input/output set verification problems [19].

## 6 Conclusions

One of the major successes of formal verification is the development of fast model checking algorithms. When talking about how improvements to model checking algorithms came about, Ken McMillan noted:

> "Engineering matters: you can't properly evaluate a technique without an efficient implementation." [18]

With this in mind, we have strived to improve the practical efficiency of the complete path-enumeration method for neural network verification. Although the geometric path-enumeration method has been proposed before, we have shown that, by a sequence of optimizations, the method's scalability can be improved by orders of magnitude.

One limitation is that we have focused on the ACAS Xu benchmarks. Although there is a risk of overfitting our optimizations to the benchmarks being considered, we believe these benchmarks are fairly general in that they contain a mix of safe and unsafe instances, where the original verification times varied

from seconds to days. In particular, we believe these networks are similar to others being used in control tasks, in terms of number of inputs and network size. Further, practical considerations prevent us from considering too many more benchmarks; our measurements already need over five days to run.

Unreported here, we were also able to run the implementation on larger perception networks to analyze $L_\infty$ perturbation properties, networks with thousands of neurons and hundreds of inputs, which succeeds when the perturbation is sufficiently small. However, we believe path enumeration is the wrong approach for those systems, as the number of paths quickly becomes too large to enumerate. Instead, overapproximation and refinement methods would likely work best, and evaluating optimizations for these methods may be done in future work. One interpretation of the results presented here is that overapproximation and refinement methods still have significant room for improvement, as it is sometimes faster to explicitly enumerate benchmarks with millions of paths.

Many of the tools we have compared against also support more complicated network structures, with different layer types and nonlinear activation functions, whereas we only focused on the subclass of networks with ReLUs and fully-connected layers. We believe that this is an important enough subclass of neural networks that the results are still meaningful. Once the neural network verification community is more mature, we expect a standard input format and a set of categorized benchmarks will arise, similar to what has happened in the SMT [2], software verification [3], and hybrid systems [1] communities.

## Acknowledgment

The material presented in this paper is based upon work supported by the National Science Foundation (NSF) under grant numbers SHF 1910017 and FMitF 1918450, the Air Force Office of Scientific Research (AFOSR) through contract numbers FA9550-18-1-0122 and FA9550-19-1-0288, the Air Force Research Laboratory (AFRL) under prime contract FA8650-15-D-2516 and the Defense Advanced Research Projects Agency (DARPA) through contract number FA8750-18-C-0089. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the AFOSR, DARPA, or NSF. Any opinions, finding, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Air Force.

## References

1. M. Althoff, S. Bak, M. Forets, G. Frehse, N. Kochdumper, R. Ray, C. Schilling, and S. Schupp. ARCH-COMP19 category report: Continuous and hybrid systems with linear continuous dynamics. In *ARCH19. 6th International Workshop on Applied Verification of Continuous and Hybrid Systems*, pages 14–40, 2019.

2. C. Barrett, A. Stump, C. Tinelli, et al. The SMT-LIB standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, volume 13, page 14, 2010.
3. D. Beyer. Competition on software verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 504–524. Springer, 2012.
4. P. S. Duggirala and M. Viswanathan. Parsimonious, simulation based verification of linear systems. In *International Conference on Computer Aided Verification*, pages 477–494. Springer, 2016.
5. S. Dutta, S. Jha, S. Sankaranarayanan, and A. Tiwari. Output range analysis for deep feedforward neural networks. In *NASA Formal Methods Symposium*, pages 121–138. Springer, 2018.
6. R. Ehlers. Formal verification of piece-wise linear feed-forward neural networks. In *International Symposium on Automated Technology for Verification and Analysis*, pages 269–286. Springer, 2017.
7. T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev. $AI^2$: Safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 3–18. IEEE, 2018.
8. A. Girard. Reachability of uncertain linear systems using zonotopes. In M. Morari and L. Thiele, editors, *Hybrid Systems: Computation and Control*, LNCS. Springer, 2005.
9. I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.
10. M. Herceg, M. Kvasnica, C. N. Jones, and M. Morari. Multi-parametric toolbox 3.0. In *2013 European control conference (ECC)*, pages 502–510. IEEE, 2013.
11. X. Huang, M. Kwiatkowska, S. Wang, and M. Wu. Safety verification of deep neural networks. In *International Conference on Computer Aided Verification*, pages 3–29. Springer, 2017.
12. K. D. Julian, J. Lopez, J. S. Brush, M. P. Owen, and M. J. Kochenderfer. Policy compression for aircraft collision avoidance systems. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, pages 1–10. IEEE, 2016.
13. G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*, pages 97–117. Springer, 2017.
14. 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*, pages 443–452. Springer, 2019.
15. C. Liu, T. Arnon, C. Lazarus, C. Barrett, and M. J. Kochenderfer. Algorithms for verifying deep neural networks. *arXiv preprint arXiv:1903.06758*, 2019.
16. A. Lomuscio and L. Maganti. An approach to reachability analysis for feed-forward relu neural networks. *arXiv preprint arXiv:1706.07351*, 2017.
17. M. Marston and G. Baca. ACAS-Xu initial self-separation flight tests. http://hdl.handle.net/2060/20150008347, 2015.
18. K. McMillan. A perspective on formal verification. David Dill @ 60 Workshop, colocated with CAV, 2017.
19. J. Mohapatra, P.-Y. Chen, S. Liu, L. Daniel, et al. Towards verifying robustness of neural networks against semantic perturbations. *arXiv preprint arXiv:1912.09533*, 2019.

20. V. R. Royo, R. Calandra, D. M. Stipanovic, and C. Tomlin. Fast neural network verification via shadow prices. *arXiv preprint arXiv:1902.07247*, 2019.
21. G. Singh, T. Gehr, M. Mirman, M. Püschel, and M. Vechev. Fast and effective robustness certification. In *Advances in Neural Information Processing Systems*, pages 10802–10813, 2018.
22. G. Singh, T. Gehr, M. Püschel, and M. Vechev. Boosting robustness certification of neural networks. *International Conference on Learning Representations (ICLR*, 2019.
23. V. Tjeng, K. Xiao, and R. Tedrake. Evaluating robustness of neural networks with mixed integer programming. *arXiv preprint arXiv:1711.07356*, 2017.
24. H.-D. Tran, D. M. Lopez, P. Musau, X. Yang, L. V. Nguyen, W. Xiang, and T. T. Johnson. Star-based reachability analysis of deep neural networks. In *International Symposium on Formal Methods*, pages 670–686. Springer, 2019.
25. H.-D. Tran, X. Yang, D. Manzanas, P. Musau, L. 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 *Proceedings of the 32nd International Conference on Computer Aided Verification*. Springer, 2020.
26. S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Efficient formal safety analysis of neural networks. In *Advances in Neural Information Processing Systems*, pages 6367–6377, 2018.
27. S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Formal security analysis of neural networks using symbolic intervals. In *27th USENIX Security Symposium*, pages 1599–1614, 2018.
28. W. Xiang, P. Musau, A. A. Wild, D. M. Lopez, N. Hamilton, X. Yang, J. Rosenfeld, and T. T. Johnson. Verification for machine learning, autonomy, and neural networks survey. *arXiv preprint arXiv:1810.01989*, 2018.
29. W. Xiang, H.-D. Tran, and T. T. Johnson. Reachable set computation and safety verification for neural networks with relu activations. *arXiv preprint arXiv:1712.08163*, 2017.
30. W. Xiang, H.-D. Tran, and T. T. Johnson. Output reachable set estimation and verification for multilayer neural networks. *IEEE transactions on neural networks and learning systems*, 29(11):5777–5783, 2018.

## A Box Bounds Algorithm for Box-Halfspace Intersection

The problem of computing the box bounds of an intersection of an initial box and a single halfspace can be computed without LP. Consider a $p$-dimensional initial box defined with lower and upper bounds $[l_1, u_1] \times \ldots \times [l_p, u_p]$. Call the constraint defining the halfspace $f\alpha \leq \mathsf{g}$, where $\alpha$ is a $p$-dimensional vector of variables, $f$ is a $p$-dimensional vector with entries $f_1, \ldots, f_p$, and $\mathsf{g}$ is a scalar.

Based on the signs of the signs of $f_1, \ldots, f_p$, we first find the vertex $v^*$ in the box that minimizes the dot product $f \cdot v^*$. This can be done by choosing the $i$th element of $v^*$ as:

$$v_i^* = \begin{cases} l_i, & \text{if } f_i \geq 0 \\ u_i, & \text{otherwise} \end{cases} \tag{6}$$

If $f \cdot v^* > \mathsf{g}$, then the intersection is the empty set. Otherwise, we attempt to contract in each of the $p$ dimensions one-by-one.

For dimension $i$, if the lower bound was used to define $v_i^*$, then we attempt to decrease $u_i$. If the upper bound was used to define $v_i^*$, then we attempt to increase $l_i$. This is done by finding the point on the edge of the box which intersects the halfspace (point $q$ in Figure 5). Without loss of generality, assume the lower bound of dimension $i$ defined $v_i^*$. The intersection point $q$ is given by $(v_1^*, v_2^*, \ldots x, \ldots v_p^*)$, where value of the $i$th coordinate, $x$, can be determined from the single-variable equation $q \cdot f = \mathsf{g}$. If $f_i$ was zero, then this equation has no solution, and we cannot contract in this dimension (the half-space and the box edge where $q$ must lie do not intersect). Otherwise, if we solve for $x$ and find $x < u_i$, then we reduce $u_i$, setting it to $x$. The process repeats for every other dimension.

## B Parallelization

The proposed approach can be parallelized in many ways. Here, we propose and evaluate a work-stealing strategy, where each thread maintains a local set of computation-state tuples and runs the high-level algorithm. Periodically, the number of tuples in each local set are communicated using a shared data structure, and if some worker thread has no work remaining, the other threads will push some of their local computation-state tuples to a shared global queue.

For this evaluation, we used the usual system setup described in the first paragraph of Section 3, which we label Laptop. In addition, to see the effect of more cores, we rented a `c5.metal` EC2 instance from Amazon Web Services, which we refer to as AWS Server. This setup ran Ubuntu 18.08, and included a dual Intel(R) Xeon(R) Platinum 8275CL processor running at 3.0 GHz, with a total of 48 physical cores (96 with hyperthreading) and 384 GB of main memory.

To evaluate parallelism, we needed to use a benchmark with sufficient difficulty where computation time dominates. For this, we chose ACAS Xu network

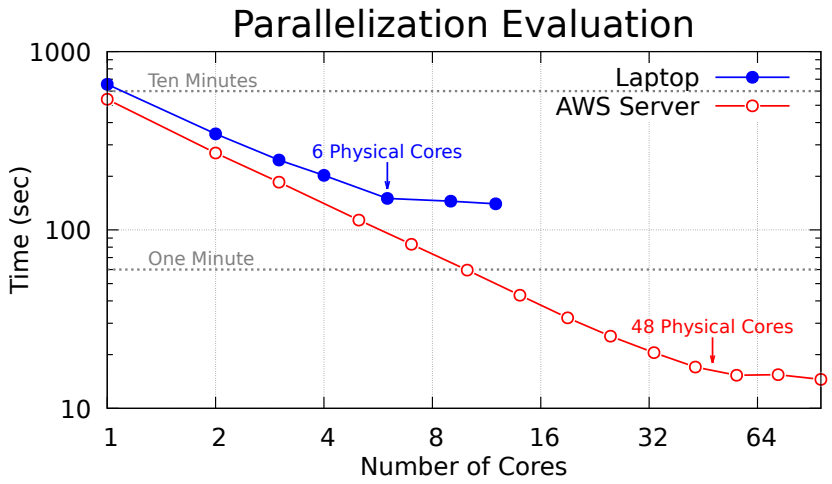


Fig. 8: Doubling the number of cores roughly halves the computation time, up to the physical core count on each platform.

4-2 with specification 2. In an earlier ACAS Xu evaluation [14], this property timed out ($> 55$ min) or ran out of memory for every tool analyzed. The single-threaded runtime on the Laptop platform with our enumeration approach was 655 seconds (about 11 minutes), which enumerated 484555 paths in the network.

An evaluation where we adjusted the number of cores available to the computation process for each of the two platforms is shown in Figure 8. The AWS Server platform was faster than the Laptop setup and, with all the cores being used, could enumerate the same 484555 paths in about 15 *seconds*. The linear trend on the log-log graph shows continuous improvement as more cores are added, up to the physical-core limit on each platform. The gains from hyperthreading are comparatively smaller. Even using all the cores, about 90% of the computation time was in the `step` function, as opposed to managing shared state. With more cores, further improvement through additional parallelization is likely possible.

**Correctness Justification:** Parallelization explores the same set of states, just in a different order.

# C Full Optimization Data

Table 2: Runtimes (sec) for each optimization. Dashes (—) are timeouts (10 min).

| Prop | Net | Result | BFS | Copy | Bound | Sim | Zono-B | Zono-S | Eager | Con-LP | Con-Sim | Par |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1-1 | SAFE | — | — | 399 | 166 | 359 | 159 | 129 | 65 | 50 | 10 |
| 1 | 1-2 | SAFE | — | — | 467 | 206 | 416 | 191 | 154 | 76 | 57 | 12 |
| 1 | 1-3 | SAFE | — | — | — | 485 | — | 496 | 375 | 197 | 163 | 32 |
| 1 | 1-4 | SAFE | — | — | — | 558 | — | 538 | 407 | 271 | 177 | 36 |
| 1 | 1-5 | SAFE | — | — | — | 492 | — | 491 | 360 | 215 | 138 | 30 |
| 1 | 1-6 | SAFE | — | — | — | — | — | — | — | — | 445 | 95 |
| 1 | 1-7 | SAFE | — | — | 539 | 250 | 518 | 259 | 190 | 113 | 78 | 17 |
| 1 | 1-8 | SAFE | — | — | — | 409 | — | 434 | 287 | 188 | 128 | 27 |
| 1 | 1-9 | SAFE | — | — | — | 476 | — | 446 | 324 | 221 | 132 | 29 |
| 1 | 2-1 | SAFE | — | — | — | — | — | — | 523 | 343 | 216 | 47 |
| 1 | 2-2 | SAFE | — | — | — | — | — | — | — | — | 599 | 119 |
| 1 | 2-3 | SAFE | — | — | — | — | — | — | 564 | 332 | 227 | 47 |
| 1 | 2-4 | SAFE | — | — | — | 383 | — | 412 | 272 | 193 | 120 | 27 |
| 1 | 2-5 | SAFE | — | — | — | — | — | — | — | — | — | 188 |
| 1 | 2-6 | SAFE | — | — | — | — | — | — | — | 517 | 400 | 82 |
| 1 | 2-7 | SAFE | — | — | — | — | — | — | — | — | — | 195 |
| 1 | 2-8 | SAFE | — | — | — | — | — | — | — | — | — | 163 |
| 1 | 2-9 | SAFE | — | — | — | — | — | — | — | — | — | 271 |
| 1 | 3-1 | SAFE | — | — | — | — | — | — | 438 | 411 | 263 | 57 |
| 1 | 3-2 | SAFE | — | — | — | — | — | — | 521 | 308 | 214 | 46 |
| 1 | 3-3 | SAFE | — | — | — | — | — | — | — | 596 | 390 | 84 |
| 1 | 3-4 | SAFE | — | — | — | 442 | — | 438 | 323 | 221 | 141 | 30 |
| 1 | 3-5 | SAFE | — | — | — | — | — | — | — | — | 401 | 86 |
| 1 | 3-6 | SAFE | — | — | — | — | — | — | — | — | — | 297 |
| 1 | 3-7 | SAFE | — | — | — | — | — | — | — | — | — | 155 |
| 1 | 3-8 | SAFE | — | — | — | — | — | — | — | — | — | 141 |
| 1 | 3-9 | SAFE | — | — | — | — | — | — | — | — | 507 | 107 |
| 1 | 4-1 | SAFE | — | — | — | — | — | — | — | — | 517 | 107 |
| 1 | 4-2 | SAFE | — | — | — | — | — | — | — | — | 568 | 124 |
| 1 | 4-3 | SAFE | — | — | — | 537 | — | 508 | 396 | 233 | 160 | 34 |
| 1 | 4-4 | SAFE | — | — | — | 523 | — | 584 | 365 | 245 | 155 | 34 |
| 1 | 4-5 | SAFE | — | — | — | — | — | — | — | — | 573 | 119 |
| 1 | 4-6 | SAFE | — | — | — | — | — | — | — | — | — | 408 |
| 1 | 4-7 | SAFE | — | — | — | — | — | — | — | — | — | 195 |
| 1 | 4-8 | SAFE | — | — | — | — | — | — | — | — | — | 131 |
| 1 | 4-9 | SAFE | — | — | — | — | — | — | — | — | — | 304 |
| 1 | 5-1 | SAFE | — | — | — | — | — | — | 482 | 322 | 232 | 48 |
| 1 | 5-2 | SAFE | — | — | — | — | — | — | — | 426 | 303 | 64 |
| 1 | 5-3 | SAFE | — | — | — | 508 | — | 498 | 366 | 214 | 143 | 32 |
| 1 | 5-4 | SAFE | — | — | — | 305 | — | 289 | 211 | 136 | 98 | 21 |
| 1 | 5-5 | SAFE | — | — | — | — | — | — | — | 368 | 264 | 57 |
| 1 | 5-6 | SAFE | — | — | — | — | — | — | — | — | — | 176 |
| 1 | 5-7 | SAFE | — | — | — | — | — | — | — | — | 474 | 97 |
| 1 | 5-8 | SAFE | — | — | — | — | — | — | — | — | — | 153 |
| 1 | 5-9 | SAFE | — | — | — | — | — | — | — | — | — | 161 |
| 2 | 1-1 | SAFE | — | — | 404 | 159 | 368 | 165 | 128 | 67 | 46 | 10 |
| 2 | 1-2 | UNSAFE | — | 58 | 24 | 11 | 23 | 12 | 9 | 5 | 4 | 1 |
| 2 | 1-3 | UNSAFE | — | 463 | 192 | 74 | 177 | 78 | 58 | 32 | 26 | 21 |
| 2 | 1-4 | UNSAFE | — | 31 | 15 | 6 | 13 | 6 | 5 | 4 | 3 | 1 |
| 2 | 1-5 | UNSAFE | — | 4 | 2 | 1 | 1 | 1 | 1 | .8 | .8 | 1 |
| 2 | 1-6 | UNSAFE | — | — | — | 517 | — | 579 | 373 | 260 | 175 | 19 |
| 2 | 1-7 | SAFE | — | — | 557 | 234 | 520 | 255 | 193 | 111 | 79 | 17 |
| 2 | 1-8 | SAFE | — | — | — | 403 | — | 399 | 297 | 184 | 126 | 27 |
| 2 | 1-9 | SAFE | — | — | — | 431 | — | 472 | 317 | 206 | 136 | 29 |
| 2 | 2-1 | UNSAFE | — | 92 | 39 | 18 | 37 | 18 | 13 | 7 | 5 | .9 |
| 2 | 2-2 | UNSAFE | — | .7 | .7 | .7 | .7 | .7 | .7 | .7 | .7 | .8 |
| 2 | 2-3 | UNSAFE | — | 8 | 4 | 2 | 4 | 2 | 2 | 1 | 1 | 1 |
| 2 | 2-4 | UNSAFE | — | 4 | 2 | 1 | 2 | 1 | 1 | 1 | .9 | .9 |
| 2 | 2-5 | UNSAFE | — | 37 | 17 | 8 | 18 | 8 | 6 | 3 | 3 | 1 |
| 2 | 2-6 | UNSAFE | — | 284 | 146 | 58 | 144 | 65 | 48 | 25 | 18 | 8 |

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 2-7 | UNSAFE | — | 506 | 250 | 85 | 256 | 96 | 78 | 43 | 30 | .9 |
| 2 | 2-8 | UNSAFE | — | 51 | 26 | 10 | 24 | 10 | 9 | 5 | 4 | 2 |
| 2 | 2-9 | UNSAFE | — | — | — | 291 | — | 320 | 242 | 132 | 94 | 2 |
| 2 | 3-1 | UNSAFE | — | 190 | 68 | 31 | 50 | 24 | 20 | 12 | 9 | 4 |
| 2 | 3-2 | UNSAFE | — | 250 | 88 | 38 | 96 | 44 | 27 | 19 | 14 | 1 |
| 2 | 3-3 | SAFE | — | — | — | — | — | — | — | 590 | 409 | 83 |
| 2 | 3-4 | UNSAFE | — | 197 | 106 | 41 | 97 | 42 | 32 | 18 | 13 | .9 |
| 2 | 3-5 | UNSAFE | — | 67 | 34 | 14 | 32 | 15 | 11 | 6 | 5 | .9 |
| 2 | 3-6 | UNSAFE | — | 27 | 10 | 5 | 11 | 5 | 5 | 3 | 2 | 5 |
| 2 | 3-7 | UNSAFE | — | 49 | 25 | 11 | 25 | 12 | 9 | 5 | 4 | 1 |
| 2 | 3-8 | UNSAFE | — | 266 | 112 | 42 | 114 | 50 | 32 | 20 | 15 | 2 |
| 2 | 3-9 | UNSAFE | — | 20 | 11 | 5 | 10 | 5 | 4 | 2 | 2 | 2 |
| 2 | 4-1 | UNSAFE | — | 115 | 45 | 19 | 40 | 20 | 14 | 8 | 7 | 5 |
| 2 | 4-2 | SAFE | — | — | — | — | — | — | — | — | 597 | 125 |
| 2 | 4-3 | UNSAFE | — | 2 | 1 | 1 | 2 | 1 | .9 | .8 | .8 | .9 |
| 2 | 4-4 | UNSAFE | — | 39 | 17 | 7 | 19 | 8 | 6 | 4 | 3 | 2 |
| 2 | 4-5 | UNSAFE | — | 470 | 239 | 97 | 200 | 94 | 71 | 34 | 27 | 2 |
| 2 | 4-6 | UNSAFE | — | 139 | 64 | 25 | 71 | 28 | 22 | 11 | 9 | 2 |
| 2 | 4-7 | UNSAFE | — | 461 | 215 | 93 | 210 | 93 | 65 | 35 | 27 | 1 |
| 2 | 4-8 | UNSAFE | — | 322 | 162 | 60 | 163 | 67 | 49 | 22 | 16 | .9 |
| 2 | 4-9 | UNSAFE | — | — | 390 | 164 | 413 | 180 | 121 | 73 | 56 | 5 |
| 2 | 5-1 | UNSAFE | — | 32 | 15 | 7 | 15 | 8 | 6 | 3 | 3 | .9 |
| 2 | 5-2 | UNSAFE | — | 91 | 39 | 18 | 30 | 16 | 12 | 6 | 6 | 1 |
| 2 | 5-3 | UNSAFE | — | — | — | 460 | — | 487 | 316 | 201 | 141 | 24 |
| 2 | 5-4 | UNSAFE | — | 2 | 1 | 1 | 1 | 1 | .9 | .8 | .8 | .9 |
| 2 | 5-5 | UNSAFE | — | 261 | 107 | 48 | 111 | 46 | 36 | 19 | 14 | 2 |
| 2 | 5-6 | UNSAFE | — | 208 | 102 | 41 | 95 | 41 | 30 | 15 | 10 | 2 |
| 2 | 5-7 | UNSAFE | — | 107 | 52 | 21 | 53 | 22 | 18 | 8 | 7 | 2 |
| 2 | 5-8 | UNSAFE | — | 302 | 161 | 63 | 160 | 67 | 50 | 27 | 19 | 1 |
| 2 | 5-9 | UNSAFE | — | — | 477 | 189 | 472 | 218 | 163 | 81 | 61 | 1 |
| 3 | 1-1 | SAFE | 561 | 526 | 232 | 116 | 125 | 80 | 58 | 103 | 58 | 12 |
| 3 | 1-2 | SAFE | 534 | 533 | 233 | 116 | 104 | 65 | 50 | 64 | 43 | 9 |
| 3 | 1-3 | SAFE | 143 | 147 | 75 | 35 | 30 | 20 | 15 | 19 | 14 | 4 |
| 3 | 1-4 | SAFE | 77 | 73 | 40 | 19 | 8 | 6 | 5 | 7 | 5 | 2 |
| 3 | 1-5 | SAFE | 88 | 84 | 42 | 21 | 10 | 7 | 6 | 8 | 6 | 2 |
| 3 | 1-6 | SAFE | 21 | 22 | 12 | 6 | 3 | 3 | 2 | 3 | 2 | 1 |
| 3 | 1-7 | UNSAFE | 8 | .7 | .7 | .7 | .7 | .7 | .7 | .7 | .7 | .8 |
| 3 | 1-8 | UNSAFE | 6 | .7 | .7 | .7 | .7 | .7 | .7 | .7 | .7 | .8 |
| 3 | 1-9 | UNSAFE | 4 | .7 | .7 | .7 | .7 | .7 | .7 | .7 | .7 | .8 |
| 3 | 2-1 | SAFE | 147 | 142 | 75 | 34 | 31 | 21 | 16 | 24 | 14 | 4 |
| 3 | 2-2 | SAFE | 59 | 55 | 30 | 14 | 12 | 8 | 6 | 10 | 6 | 2 |
| 3 | 2-3 | SAFE | 108 | 101 | 50 | 25 | 19 | 12 | 9 | 14 | 9 | 3 |
| 3 | 2-4 | SAFE | 6 | 6 | 4 | 2 | 1 | 1 | 1 | 1 | 1 | 1 |
| 3 | 2-5 | SAFE | 33 | 33 | 18 | 9 | 4 | 4 | 3 | 4 | 3 | 1 |
| 3 | 2-6 | SAFE | 5 | 5 | 4 | 2 | 1 | 1 | 1 | 1 | .9 | 1 |
| 3 | 2-7 | SAFE | 17 | 16 | 11 | 5 | 3 | 2 | 2 | 2 | 2 | 1 |
| 3 | 2-8 | SAFE | 6 | 6 | 5 | 2 | 1 | 1 | 1 | 1 | 1 | 1 |
| 3 | 2-9 | SAFE | 4 | 4 | 3 | 2 | .9 | .9 | .8 | 1 | .9 | .9 |
| 3 | 3-1 | SAFE | 57 | 53 | 25 | 12 | 11 | 7 | 5 | 9 | 6 | 2 |
| 3 | 3-2 | SAFE | 578 | 537 | 226 | 117 | 93 | 53 | 40 | 59 | 36 | 8 |
| 3 | 3-3 | SAFE | 128 | 128 | 65 | 31 | 22 | 14 | 11 | 13 | 11 | 3 |
| 3 | 3-4 | SAFE | 27 | 26 | 16 | 7 | 5 | 4 | 3 | 4 | 2 | 1 |
| 3 | 3-5 | SAFE | 16 | 16 | 10 | 5 | 2 | 2 | 2 | 2 | 2 | 1 |
| 3 | 3-6 | SAFE | 31 | 33 | 20 | 10 | 5 | 4 | 3 | 3 | 3 | 1 |
| 3 | 3-7 | SAFE | 2 | 2 | 2 | 1 | .8 | .8 | .7 | .8 | .8 | .8 |
| 3 | 3-8 | SAFE | 12 | 12 | 8 | 4 | 2 | 2 | 1 | 2 | 1 | 1 |
| 3 | 3-9 | SAFE | 16 | 15 | 10 | 5 | 3 | 2 | 2 | 2 | 2 | 1 |
| 3 | 4-1 | SAFE | 18 | 18 | 11 | 5 | 5 | 3 | 2 | 4 | 3 | 1 |
| 3 | 4-2 | SAFE | 189 | 187 | 88 | 43 | 44 | 24 | 19 | 25 | 16 | 4 |
| 3 | 4-3 | SAFE | 282 | 283 | 136 | 63 | 64 | 35 | 29 | 32 | 24 | 5 |
| 3 | 4-4 | SAFE | 12 | 11 | 7 | 4 | 2 | 1 | 1 | 2 | 1 | 1 |
| 3 | 4-5 | SAFE | 4 | 4 | 3 | 2 | 1 | 1 | .9 | 1 | .9 | 1 |
| 3 | 4-6 | SAFE | 33 | 34 | 20 | 10 | 7 | 5 | 4 | 4 | 3 | 1 |
| 3 | 4-7 | SAFE | 15 | 15 | 11 | 5 | 2 | 2 | 2 | 2 | 2 | 1 |
| 3 | 4-8 | SAFE | 11 | 12 | 8 | 4 | 2 | 1 | 1 | 2 | 1 | 1 |
| 3 | 4-9 | SAFE | 12 | 11 | 8 | 4 | 2 | 2 | 2 | 2 | 1 | 1 |
| 3 | 5-1 | SAFE | 97 | 91 | 50 | 25 | 19 | 12 | 9 | 14 | 9 | 3 |
| 3 | 5-2 | SAFE | 18 | 19 | 11 | 6 | 5 | 3 | 2 | 4 | 2 | 1 |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 5-3 | SAFE | 22 | 23 | 12 | 6 | 5 | 3 | 3 | 4 | 3 | 1 |
| 3 | 5-4 | SAFE | 11 | 11 | 7 | 4 | 2 | 2 | 1 | 2 | 1 | 1 |
| 3 | 5-5 | SAFE | 15 | 14 | 10 | 5 | 2 | 2 | 2 | 2 | 2 | 1 |
| 3 | 5-6 | SAFE | 23 | 21 | 14 | 7 | 3 | 3 | 2 | 3 | 2 | 1 |
| 3 | 5-7 | SAFE | 2 | 2 | 2 | 1 | .8 | .8 | .7 | .8 | .7 | .8 |
| 3 | 5-8 | SAFE | 37 | 38 | 24 | 10 | 6 | 4 | 4 | 5 | 3 | 1 |
| 3 | 5-9 | SAFE | 2 | 2 | 2 | 1 | .9 | .8 | .7 | .8 | .8 | .8 |
| 4 | 1-1 | SAFE | 149 | 150 | 72 | 34 | 33 | 22 | 16 | 23 | 16 | 4 |
| 4 | 1-2 | SAFE | 135 | 130 | 52 | 27 | 21 | 15 | 12 | 16 | 11 | 3 |
| 4 | 1-3 | SAFE | 95 | 96 | 44 | 23 | 18 | 12 | 10 | 13 | 9 | 3 |
| 4 | 1-4 | SAFE | 12 | 11 | 7 | 4 | 2 | 2 | 2 | 2 | 2 | 1 |
| 4 | 1-5 | SAFE | 81 | 84 | 42 | 20 | 12 | 9 | 8 | 9 | 7 | 2 |
| 4 | 1-6 | SAFE | 41 | 37 | 20 | 11 | 7 | 5 | 4 | 6 | 4 | 2 |
| 4 | 1-7 | UNSAFE | 6 | .7 | .7 | .7 | .7 | .7 | .7 | .7 | .7 | .8 |
| 4 | 1-8 | UNSAFE | 7 | .8 | .7 | .7 | .7 | .7 | .7 | .7 | .7 | .8 |
| 4 | 1-9 | UNSAFE | 5 | .7 | .7 | .7 | .7 | .7 | .7 | .7 | .7 | .8 |
| 4 | 2-1 | SAFE | 38 | 41 | 21 | 11 | 7 | 5 | 5 | 7 | 4 | 2 |
| 4 | 2-2 | SAFE | 50 | 51 | 27 | 13 | 8 | 6 | 5 | 6 | 4 | 2 |
| 4 | 2-3 | SAFE | 9 | 9 | 6 | 3 | 2 | 2 | 2 | 2 | 1 | 1 |
| 4 | 2-4 | SAFE | 8 | 9 | 5 | 3 | 2 | 2 | 1 | 2 | 1 | 1 |
| 4 | 2-5 | SAFE | 28 | 27 | 14 | 7 | 6 | 4 | 4 | 4 | 3 | 1 |
| 4 | 2-6 | SAFE | 15 | 15 | 9 | 5 | 3 | 2 | 2 | 2 | 2 | 1 |
| 4 | 2-7 | SAFE | 7 | 7 | 5 | 3 | 1 | 1 | 1 | 1 | 1 | 1 |
| 4 | 2-8 | SAFE | 40 | 43 | 25 | 11 | 5 | 4 | 3 | 4 | 3 | 1 |
| 4 | 2-9 | SAFE | 3 | 3 | 3 | 2 | .9 | .9 | .9 | .9 | .9 | .9 |
| 4 | 3-1 | SAFE | 56 | 52 | 27 | 13 | 7 | 6 | 5 | 6 | 5 | 2 |
| 4 | 3-2 | SAFE | 63 | 61 | 31 | 15 | 12 | 9 | 7 | 11 | 7 | 2 |
| 4 | 3-3 | SAFE | 10 | 9 | 6 | 3 | 2 | 2 | 2 | 2 | 2 | 1 |
| 4 | 3-4 | SAFE | 12 | 12 | 7 | 3 | 2 | 2 | 2 | 2 | 2 | 1 |
| 4 | 3-5 | SAFE | 38 | 40 | 22 | 10 | 8 | 6 | 4 | 5 | 4 | 2 |
| 4 | 3-6 | SAFE | 20 | 20 | 12 | 6 | 3 | 3 | 2 | 3 | 2 | 1 |
| 4 | 3-7 | SAFE | 17 | 17 | 11 | 5 | 3 | 2 | 2 | 2 | 2 | 1 |
| 4 | 3-8 | SAFE | 7 | 7 | 5 | 2 | 2 | 2 | 1 | 1 | 1 | 1 |
| 4 | 3-9 | SAFE | 51 | 48 | 29 | 13 | 7 | 5 | 5 | 5 | 4 | 2 |
| 4 | 4-1 | SAFE | 7 | 7 | 5 | 3 | 2 | 1 | 1 | 2 | 1 | 1 |
| 4 | 4-2 | SAFE | 14 | 14 | 8 | 5 | 3 | 2 | 2 | 2 | 2 | 1 |
| 4 | 4-3 | SAFE | 26 | 27 | 14 | 8 | 5 | 4 | 3 | 5 | 3 | 1 |
| 4 | 4-4 | SAFE | 20 | 20 | 11 | 6 | 3 | 2 | 2 | 2 | 2 | 1 |
| 4 | 4-5 | SAFE | 17 | 16 | 9 | 5 | 3 | 2 | 2 | 2 | 2 | 1 |
| 4 | 4-6 | SAFE | 30 | 30 | 15 | 7 | 5 | 3 | 3 | 4 | 3 | 1 |
| 4 | 4-7 | SAFE | 3 | 3 | 2 | 1 | 1 | .9 | .9 | .9 | .8 | .8 |
| 4 | 4-8 | SAFE | 24 | 23 | 16 | 7 | 4 | 3 | 2 | 3 | 2 | 1 |
| 4 | 4-9 | SAFE | 43 | 40 | 24 | 12 | 5 | 4 | 4 | 4 | 4 | 2 |
| 4 | 5-1 | SAFE | 57 | 53 | 26 | 14 | 10 | 7 | 6 | 8 | 5 | 2 |
| 4 | 5-2 | SAFE | 38 | 34 | 17 | 9 | 7 | 4 | 4 | 5 | 4 | 2 |
| 4 | 5-3 | SAFE | 14 | 13 | 8 | 4 | 3 | 2 | 2 | 3 | 2 | 1 |
| 4 | 5-4 | SAFE | 13 | 13 | 8 | 4 | 2 | 2 | 2 | 2 | 2 | 1 |
| 4 | 5-5 | SAFE | 17 | 17 | 11 | 6 | 3 | 3 | 2 | 2 | 2 | 1 |
| 4 | 5-6 | SAFE | 10 | 10 | 6 | 3 | 2 | 2 | 2 | 2 | 1 | 1 |
| 4 | 5-7 | SAFE | 3 | 3 | 2 | 1 | .9 | .8 | .8 | .9 | .8 | .8 |
| 4 | 5-8 | SAFE | 8 | 8 | 6 | 3 | 2 | 1 | 1 | 1 | 1 | 1 |
| 4 | 5-9 | SAFE | 14 | 13 | 8 | 4 | 2 | 2 | 2 | 2 | 2 | 1 |

# D Full Tool Comparison Data

This section contains the complete data measured in the optimization improvements from Section 3.

Table 3: Runtimes (sec) for each tool. Dashes (—) are timeouts (10 min).

| Prop | Net | Result | Marabou | NNV Exact | Our Method | Neurify |
|---|---|---|---|---|---|---|
| 1 | 1-1 | SAFE | 95 | — | 10 | **.1** |
| 1 | 1-2 | SAFE | 168 | — | 12 | **.2** |
| 1 | 1-3 | SAFE | — | — | 32 | **1** |
| 1 | 1-4 | SAFE | — | — | 36 | **2** |
| 1 | 1-5 | SAFE | 119 | — | 30 | **.2** |
| 1 | 1-6 | SAFE | 110 | — | 95 | **.2** |
| 1 | 1-7 | SAFE | 63 | — | 17 | **.1** |
| 1 | 1-8 | SAFE | 56 | — | 27 | **.1** |
| 1 | 1-9 | SAFE | 43 | — | 29 | **.1** |
| 1 | 2-1 | SAFE | — | — | 47 | **.6** |
| 1 | 2-2 | SAFE | — | — | 119 | **1** |
| 1 | 2-3 | SAFE | — | — | 47 | **1** |
| 1 | 2-4 | SAFE | 294 | — | 27 | **.5** |
| 1 | 2-5 | SAFE | — | — | 188 | **4** |
| 1 | 2-6 | SAFE | — | — | 82 | **3** |
| 1 | 2-7 | SAFE | — | — | 195 | **11** |
| 1 | 2-8 | SAFE | — | — | 163 | **3** |
| 1 | 2-9 | SAFE | — | — | 271 | **8** |
| 1 | 3-1 | SAFE | — | — | 57 | **.4** |
| 1 | 3-2 | SAFE | — | — | 46 | **.7** |
| 1 | 3-3 | SAFE | 521 | — | 84 | **1** |
| 1 | 3-4 | SAFE | 510 | — | 30 | **.6** |
| 1 | 3-5 | SAFE | — | — | 86 | **2** |
| 1 | 3-6 | SAFE | — | — | 297 | **28** |
| 1 | 3-7 | SAFE | — | — | 155 | **12** |
| 1 | 3-8 | SAFE | — | — | 141 | **8** |
| 1 | 3-9 | SAFE | — | — | 107 | **11** |
| 1 | 4-1 | SAFE | — | — | 107 | **16** |
| 1 | 4-2 | SAFE | — | — | 124 | **3** |
| 1 | 4-3 | SAFE | — | — | 34 | **1** |
| 1 | 4-4 | SAFE | 387 | — | 34 | **.7** |
| 1 | 4-5 | SAFE | — | — | 119 | **3** |
| 1 | 4-6 | SAFE | — | — | 408 | **22** |
| 1 | 4-7 | SAFE | — | — | 195 | **23** |
| 1 | 4-8 | SAFE | — | — | 131 | **47** |
| 1 | 4-9 | SAFE | — | — | 304 | **21** |
| 1 | 5-1 | SAFE | 353 | — | 48 | **.4** |
| 1 | 5-2 | SAFE | 522 | — | 64 | **.7** |
| 1 | 5-3 | SAFE | 128 | — | 32 | **.2** |
| 1 | 5-4 | SAFE | 574 | — | 21 | **.4** |
| 1 | 5-5 | SAFE | — | — | 57 | **1** |
| 1 | 5-6 | SAFE | — | — | 176 | **15** |
| 1 | 5-7 | SAFE | — | — | 97 | **3** |
| 1 | 5-8 | SAFE | — | — | 153 | **16** |
| 1 | 5-9 | SAFE | — | — | 161 | **8** |
| 2 | 1-1 | SAFE | — | — | 10 | **.6** |
| 2 | 1-2 | UNSAFE | 254 | — | **1** | 3 |
| 2 | 1-3 | UNSAFE | — | — | 21 | **11** |
| 2 | 1-4 | UNSAFE | — | — | **1** | 10 |
| 2 | 1-5 | UNSAFE | — | — | **1** | — |
| 2 | 1-6 | UNSAFE | — | — | **19** | 52 |
| 2 | 1-7 | SAFE | — | — | 17 | **6** |
| 2 | 1-8 | SAFE | — | — | 27 | **23** |
| 2 | 1-9 | SAFE | — | — | 29 | **11** |
| 2 | 2-1 | UNSAFE | 59 | — | .9 | **.1** |

| | | | | | | |
|---|---|---|---|---|---|---|
| 2 | 2-2 | UNSAFE | — | — | .8 | **.1** |
| 2 | 2-3 | UNSAFE | 549 | — | 1 | **.1** |
| 2 | 2-4 | UNSAFE | 18 | — | .9 | **.1** |
| 2 | 2-5 | UNSAFE | 547 | — | 1 | **.1** |
| 2 | 2-6 | UNSAFE | — | — | 8 | **.1** |
| 2 | 2-7 | UNSAFE | 24 | — | .9 | **.1** |
| 2 | 2-8 | UNSAFE | 102 | — | 2 | **.1** |
| 2 | 2-9 | UNSAFE | — | — | **2** | — |
| 2 | 3-1 | UNSAFE | 97 | — | 4 | **.1** |
| 2 | 3-2 | UNSAFE | 345 | — | **1** | — |
| 2 | 3-3 | SAFE | — | — | **83** | — |
| 2 | 3-4 | UNSAFE | — | — | .9 | **.1** |
| 2 | 3-5 | UNSAFE | 319 | — | .9 | **.1** |
| 2 | 3-6 | UNSAFE | 471 | — | 5 | **.1** |
| 2 | 3-7 | UNSAFE | — | — | **1** | — |
| 2 | 3-8 | UNSAFE | — | — | 2 | **.1** |
| 2 | 3-9 | UNSAFE | 457 | — | 2 | **.1** |
| 2 | 4-1 | UNSAFE | — | — | 5 | **.2** |
| 2 | 4-2 | SAFE | — | — | **125** | — |
| 2 | 4-3 | UNSAFE | 566 | — | .9 | **.1** |
| 2 | 4-4 | UNSAFE | 288 | — | 2 | **.1** |
| 2 | 4-5 | UNSAFE | — | — | 2 | **.1** |
| 2 | 4-6 | UNSAFE | 419 | — | 2 | **.1** |
| 2 | 4-7 | UNSAFE | — | — | 1 | **.1** |
| 2 | 4-8 | UNSAFE | 336 | — | .9 | **.1** |
| 2 | 4-9 | UNSAFE | — | — | **5** | 45 |
| 2 | 5-1 | UNSAFE | 119 | — | .9 | **.1** |
| 2 | 5-2 | UNSAFE | 24 | — | 1 | **.1** |
| 2 | 5-3 | UNSAFE | — | — | **24** | — |
| 2 | 5-4 | UNSAFE | 360 | — | .9 | **.1** |
| 2 | 5-5 | UNSAFE | 278 | — | 2 | **.1** |
| 2 | 5-6 | UNSAFE | 547 | — | 2 | **.1** |
| 2 | 5-7 | UNSAFE | 17 | — | 2 | **.1** |
| 2 | 5-8 | UNSAFE | 246 | — | 1 | **.1** |
| 2 | 5-9 | UNSAFE | 47 | — | 1 | **.1** |
| 3 | 1-1 | SAFE | — | 564 | **12** | 104 |
| 3 | 1-2 | SAFE | — | 283 | 9 | **2** |
| 3 | 1-3 | SAFE | — | 58 | 4 | **3** |
| 3 | 1-4 | SAFE | 342 | 12 | 2 | **.3** |
| 3 | 1-5 | SAFE | 520 | 17 | 2 | **.2** |
| 3 | 1-6 | SAFE | 43 | 4 | 1 | **.1** |
| 3 | 1-7 | UNSAFE | 12 | 2 | .8 | **.1** |
| 3 | 1-8 | UNSAFE | 12 | 2 | .8 | **.1** |
| 3 | 1-9 | UNSAFE | 12 | 1 | .8 | **.05** |
| 3 | 2-1 | SAFE | — | 70 | **4** | 21 |
| 3 | 2-2 | SAFE | — | 23 | **2** | 8 |
| 3 | 2-3 | SAFE | — | 39 | **3** | 3 |
| 3 | 2-4 | SAFE | 15 | 2 | 1 | **.5** |
| 3 | 2-5 | SAFE | 18 | 7 | 1 | **.4** |
| 3 | 2-6 | SAFE | 15 | 1 | 1 | **.04** |
| 3 | 2-7 | SAFE | 16 | 4 | 1 | **.3** |
| 3 | 2-8 | SAFE | 15 | 2 | 1 | **.1** |
| 3 | 2-9 | SAFE | 13 | 1 | .9 | **.03** |
| 3 | 3-1 | SAFE | 406 | 21 | **2** | 3 |
| 3 | 3-2 | SAFE | — | 247 | 8 | **6** |
| 3 | 3-3 | SAFE | — | 35 | 3 | **.2** |
| 3 | 3-4 | SAFE | 47 | 8 | 1 | **.4** |
| 3 | 3-5 | SAFE | 15 | 4 | **1** | 5 |
| 3 | 3-6 | SAFE | 390 | 7 | **1** | 151 |
| 3 | 3-7 | SAFE | 13 | .9 | .8 | **.1** |
| 3 | 3-8 | SAFE | 36 | 3 | **1** | 4 |
| 3 | 3-9 | SAFE | 45 | 4 | **1** | 3 |
| 3 | 4-1 | SAFE | — | 8 | **1** | 8 |
| 3 | 4-2 | SAFE | — | 88 | **4** | 97 |
| 3 | 4-3 | SAFE | — | 130 | 5 | **2** |
| 3 | 4-4 | SAFE | 14 | 2 | 1 | **.1** |
| 3 | 4-5 | SAFE | 14 | 1 | 1 | **.1** |
| 3 | 4-6 | SAFE | 102 | 11 | 1 | **.2** |

| | | | | | | |
|---|---|---|---|---|---|---|
| 3 | 4-7 | SAFE | 96 | 3 | 1 | **.6** |
| 3 | 4-8 | SAFE | 85 | 2 | **1** | 2 |
| 3 | 4-9 | SAFE | 33 | 3 | 1 | **.1** |
| 3 | 5-1 | SAFE | — | 35 | **3** | 21 |
| 3 | 5-2 | SAFE | — | 8 | **1** | 2 |
| 3 | 5-3 | SAFE | 146 | 8 | 1 | **.2** |
| 3 | 5-4 | SAFE | 17 | 3 | 1 | **.2** |
| 3 | 5-5 | SAFE | 24 | 4 | 1 | **.5** |
| 3 | 5-6 | SAFE | 88 | 5 | 1 | **.9** |
| 3 | 5-7 | SAFE | 14 | .6 | .8 | **.04** |
| 3 | 5-8 | SAFE | 43 | 9 | 1 | **.1** |
| 3 | 5-9 | SAFE | 14 | .9 | .8 | **.1** |
| 4 | 1-1 | SAFE | — | 82 | 4 | **1** |
| 4 | 1-2 | SAFE | — | 50 | 3 | **1** |
| 4 | 1-3 | SAFE | — | 36 | 3 | **.4** |
| 4 | 1-4 | SAFE | 105 | 3 | 1 | **.2** |
| 4 | 1-5 | SAFE | 504 | 24 | 2 | **.4** |
| 4 | 1-6 | SAFE | 89 | 12 | 2 | **.2** |
| 4 | 1-7 | UNSAFE | 12 | 2 | .8 | **.1** |
| 4 | 1-8 | UNSAFE | 12 | 2 | .8 | **.1** |
| 4 | 1-9 | UNSAFE | 12 | 2 | .8 | **.1** |
| 4 | 2-1 | SAFE | 171 | 14 | 2 | **.8** |
| 4 | 2-2 | SAFE | 520 | 14 | **2** | 2 |
| 4 | 2-3 | SAFE | 77 | 3 | 1 | **.8** |
| 4 | 2-4 | SAFE | 23 | 3 | 1 | **.2** |
| 4 | 2-5 | SAFE | 61 | 11 | 1 | **.4** |
| 4 | 2-6 | SAFE | 90 | 5 | 1 | **.3** |
| 4 | 2-7 | SAFE | 14 | 2 | 1 | **.1** |
| 4 | 2-8 | SAFE | 43 | 8 | 1 | **.1** |
| 4 | 2-9 | SAFE | 13 | 1 | .9 | **.03** |
| 4 | 3-1 | SAFE | — | 13 | 2 | **1** |
| 4 | 3-2 | SAFE | 134 | 27 | 2 | **.4** |
| 4 | 3-3 | SAFE | 21 | 4 | 1 | **.1** |
| 4 | 3-4 | SAFE | 20 | 4 | 1 | **.2** |
| 4 | 3-5 | SAFE | 59 | 15 | 2 | **1** |
| 4 | 3-6 | SAFE | 66 | 5 | **1** | 2 |
| 4 | 3-7 | SAFE | 16 | 4 | 1 | **.3** |
| 4 | 3-8 | SAFE | 29 | 3 | 1 | **.3** |
| 4 | 3-9 | SAFE | 63 | 12 | 2 | **1** |
| 4 | 4-1 | SAFE | 78 | 3 | **1** | 3 |
| 4 | 4-2 | SAFE | 60 | 5 | **1** | 2 |
| 4 | 4-3 | SAFE | 134 | 10 | 1 | **1** |
| 4 | 4-4 | SAFE | 41 | 5 | **1** | 1 |
| 4 | 4-5 | SAFE | 62 | 4 | **1** | 2 |
| 4 | 4-6 | SAFE | 14 | 8 | 1 | **.04** |
| 4 | 4-7 | SAFE | 21 | 1 | .8 | **.2** |
| 4 | 4-8 | SAFE | 37 | 6 | 1 | **.2** |
| 4 | 4-9 | SAFE | 25 | 8 | 2 | **.1** |
| 4 | 5-1 | SAFE | 339 | 19 | **2** | 3 |
| 4 | 5-2 | SAFE | 51 | 12 | 2 | **.5** |
| 4 | 5-3 | SAFE | 52 | 5 | 1 | **.2** |
| 4 | 5-4 | SAFE | 31 | 4 | 1 | **.2** |
| 4 | 5-5 | SAFE | 49 | 5 | 1 | **.6** |
| 4 | 5-6 | SAFE | 76 | 3 | 1 | **.3** |
| 4 | 5-7 | SAFE | 14 | 1 | .8 | **.04** |
| 4 | 5-8 | SAFE | 31 | 3 | 1 | **.1** |
| 4 | 5-9 | SAFE | 26 | 3 | 1 | **.1** |