PICID: Proof-Driven Clause Learning in Neural Network Verification

Omri Isac[1§], Idan Refaeli[1§], Haoze Wu[2], Clark Barrett[3], Guy Katz[1]
[1]The Hebrew University of Jerusalem, Jerusalem, Israel [2]Amherst College, Amherst, Massachusetts, USA
[3]Stanford University, Stanford, California, USA
{omri.isac, idan.refaeli, guy.katz}@mail.huji.ac.il, hwu@amherst.edu, barrettc@stanford.edu

***Abstract*—Current Deep Neural Network (DNN) verifiers are typically designed to prioritize scalability over reliability. Reliability can be reinforced through the generation of *proofs* that are checkable by trusted, external proof checkers. To date, only a handful of verifiers support proof production; and these rely on verifier-specific formats, and balance between scalability, proof detail, and the trustworthiness of their proof checker. In this tool paper, we introduce PICID, a DNN verifier that produces proofs in the standard Alethe format for SMT solving, checkable by an independent checker. PICID implements a parallel CDCL($\mathcal{T}$) architecture that integrates the state-of-the-art, proof-producing CaDiCaL SAT solver with the Marabou DNN verifier. Furthermore, PICID leverages `UNSAT` proofs to derive conflict clauses. Our evaluation shows that PICID generates valid proofs in the vast majority of cases and significantly outperforms existing tools that produce comparable proofs.**

I. INTRODUCTION

Deep Neural Networks (DNNs) have achieved remarkable success across many computational tasks, yet their internal structure and parameters are largely unintelligible, which undermines trust in their decisions. To address this problem, the formal verification community has developed multiple algorithms for *DNN verification*, which can provably determine whether a DNN complies with given specifications [1], [4], [5], [7], [16], [16], [26], [33], [34], [38], [46], [47], [50], [53], [55], [56], [61], [63], [66], [70], [73].

DNN verifiers are typically designed with an emphasis on performance and scalability [46], which is essential for handling real-world DNNs [29], [46], [54]. However, the *reliability* of these verifiers has received lesser attention. In practice, many DNN verifiers rely on floating-point arithmetic for internal computations, which can be numerically unstable and may compromise soundness [42], [60], [75]. Indeed, even state-of-the-art verifiers occasionally produce incorrect results on some queries [15], [46]. Altogether, these issues undermine the reliability of the verification process as a whole.

The mainstream approach for addressing this challenge is through *proof production*: the generation of checkable, mathematical artifacts that accompany verifier results. This is common practice, e.g., in the SAT and SMT communities [10], [37]. Proof artifacts can be checked by a reliable proof checker that either certifies the verification result, or exposes errors in the verification process. To date, only a handful of DNN verifiers support proof production [25], [39], and these suffer from several limitations: (i) These proofs have varying levels of detail: some of them are fairly high-level, omitting many details and placing a computational burden on the checker; (ii) The proofs are often written in solver-specific formats, making it difficult, e.g., to apply multiple checkers or to compare proofs produced by different tools; and (iii) The proof checkers used are often ad-hoc tools, which may themselves contain errors. To the best of our knowledge, only a single proof checker for DNN verification proofs has been formally verified [23], [24]; however, this checker lacks support for basic optimizations applied by DNN verifiers, thus limiting the scalability of proof-producing verifiers to small-scale DNNs.

Here, we present a new tool that seeks to begin addressing these limitations. Our goal is to improve the scalability of *reliable* DNN verifiers, namely DNN verifiers that produce proofs in a standard format, checkable by reliable proof checkers. To this end, we introduce PICID, which builds on the proof-producing DNN verifier Marabou [39], [68]. PICID invokes Marabou as a theory-solver ($\mathcal{T}$-solver), integrating it with the CaDiCaL SAT solver [14] within a parallel CDCL($\mathcal{T}$) architecture. For `UNSAT` queries, it reduces both Marabou proofs and CaDiCaL's LRAT proofs [21], [52] into the standard Alethe format for SMT proofs [9], [57], and merges them to a complete proof of unsatisfiability. In particular, PICID introduces the following novel features:

1) A CDCL($\mathcal{T}$) architecture for a DNN verifier, employing the state-of-the-art SAT solver CaDiCaL [14], via the standard IPASIR-UP interface [30], [31]. Conflict clauses are derived based on `UNSAT` proofs, using a variant of the proof minimization algorithm presented in [41].
2) Production of `UNSAT` proofs in the standard Alethe format for SMT solvers, checkable by its standard proof checker [3], and reconstructible in a verified environment [20], [28], [48].
3) A *Split and Conquer* (SNC) parallelization technique [69], which produces proofs for each worker thread, and then combines them into an overall proof for the query at hand.

Our evaluation indicates that PICID significantly outperforms an existing DNN verifier and an existing SMT solver that produce Alethe proofs. Furthermore, over 99% of the proofs generated by PICID successfully passed their checks. For the single query for which PICID returned an unsound result,

§Both authors contributed equally.

https://doi.org/

the proof mechanism indicated the potential steps in the verification process that could have caused the error. By that, PICID pushes forward the scalability of DNN verifiers, while affording high levels of reliability compared to existing DNN verifiers.

The paper is organized as follows: In Section II we provide relevant background on DNN verification and the Alethe proof format. Section III includes an overview description of PICID. Then, we explain its CDCL($\mathcal{T}$), proofs, and parallelization modules in Sections IV, V and VI, respectively. Section VII describes our experimental evaluation, and we conclude and discuss related and future work in Section VIII.

II. BACKGROUND

In this section, we provide relevant background on DNN verification and the Alethe proof format used by PICID. We assume the reader is familiar with the basics of DPLL($\mathcal{T}$) [32].

A. DNN Verification

Deep Neural Networks. [36] Formally, a DNN $\mathcal{N} : \mathbb{R}^n \to \mathbb{R}^m$ is a directed and weighted graph, structured as a sequence of k layers $L_0, ..., L_{k-1}$. Each layer L_i consists of $s_i \in \mathbb{N}$ nodes $v_i^1, ..., v_i^{s_i}$ and s_i biases $b_i^j \in \mathbb{Q}$. Each directed edge in the DNN connects a node from some layer to a node in the subsequent layer, i.e., is of the form (v_{i-1}^l, v_i^j). Furthermore, each edge is labeled with a weight $w_{i,j,l} \in \mathbb{Q}$.

The assignment to the nodes in the first, input layer is defined by $v_0^j = x_j$, where $\overline{x} \in \mathbb{R}^n$ is the input vector. The assignment for the j^{th} node in the $1 < i < k$ layer is computed as $v_i^j = f_i\left(\sum_{l=1}^{s_{i-1}} w_{i,j,l} \cdot v_{i-1}^l + b_i^j\right)$ for some non-linear activation function $f_i : \mathbb{R} \to \mathbb{R}$. Neurons in the final, output layer are evaluated similarly but without applying an activation function.

In this work, we focus on DNNs with the ReLU activation function, defined as $\text{ReLU}(x) := \max(x, 0)$. Note that ReLU has two linear phases: For non-negative inputs, the functions is the identity and regarded as *active*; otherwise, the function is constant zero, and regarded as *inactive*.

Example 1: A DNN with three layers appears in Figure 1. For simplicity, all biases are set to 0 and are omitted. Given the input $[0, 1, 1]$ the value of v_1, v_2 is computed by $v_1 = \text{ReLU}(0 - 1) = 0$ and $v_2 = \text{ReLU}(2 + 1) = 3$, thus the network's output is $0 + 3 = 3$.

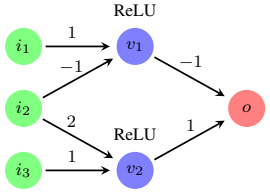

Fig. 1: A toy DNN with ReLU activations.

DNN Verification. Consider a DNN $\mathcal{N} : \mathbb{R}^k \to \mathbb{R}^m$ where $i \in \mathbb{R}^k, o \in \mathbb{R}^m$ denote the network's inputs and outputs. The *DNN verification problem* is the problem of deciding whether there exist $(i, o) \in \mathbb{R}^{k+m}$ such that $\varphi(i) \wedge (\mathcal{N}(i) = o) \wedge \psi(o)$ for some input and output properties φ, ψ. If such vectors exist, the problem is satisfiable and denoted SAT. Otherwise, it is unsatisfiable and denoted UNSAT. As common in the literature, we consider *quantifier-free linear properties*, i.e., formulas without quantifiers whose atoms are linear equations and inequalities.

Example 2: Consider the DNN from Figure 1 and the properties $\varphi(i) := \bigwedge_{j \in 1,2,3} 0 \leq i_j \leq 1$, $\psi(o) := o \leq -2$. This verification problem is UNSAT. Alternatively, for the same input property φ, the same DNN, and output property $\psi'(o) := o \leq 4$, the instance is SAT, as demonstrated by the input in Example 1.

A *DNN verification query* is encoded as a tuple $Q = \langle x, A, l, u, R \rangle$, where $\langle x, A, l, u \rangle$ constitutes a linear program [22], [44]. Formally, let $x = [x_1, \ldots, x_n]^\intercal \in \mathbb{R}^n$ denote variables, $A \in M_{m \times n}(\mathbb{R})$ denote a constraint matrix and $l, u \in (\mathbb{R} \cup \{\pm\infty\})^n$ denote vectors of bounds. Throughout the paper, we use $l(x_j)$ and $u(x_j)$, to refer to the lower and upper bounds (respectively) of x_j, and denote $l \leq u$ to indicate that for each element j, $l(x_j) \leq u(x_j)$.

R represents the set of ReLU activation constraints of the form $R_j := f_j = \text{ReLU}(b_j)$, for variables $b_j, f_j \in V$. In addition, a fresh auxiliary variable a_j is added per ReLU constraint, representing the non-negative difference of their output and input. The variable is added to x, with an equation $a_j = f_j - b_j$ added to A, and the bounds $l(a_j) = 0$, $u(a_j) = u(f_j) - l(b_j)$ added to l and u, respectively. The DNN verification problem is then to decide the satisfiability of $(A \cdot x = 0) \wedge (l \leq x \leq u) \wedge \bigwedge_{R_j \in R} f_j = \text{ReLU}(b_j)$

Example 3: We demonstrate this encoding with the example presented in Figure 1 and the property from Example 2. Denote i_1, i_2, i_3 and o the network's inputs and output, and denote b_j, f_j ($j \in \{1, 2\}$) the input and output of neuron v_i, which yield the constraints R_i respectively. The affine transformations of the network are reduced to the constraint matrix $A \cdot x = \overline{0}$:

$$\begin{aligned} i_1 - i_2 - b_1 &= 0 \\ 2i_2 + i_3 - b_2 &= 0 \\ f_2 - f_1 - o &= 0 \\ f_1 - b_1 - a_1 &= 0 \\ f_2 - b_2 - a_2 &= 0 \end{aligned}$$

The property is encoded as the inequalities in Example 2, where we add the inequalities $(0 \leq f_j)$ (for $j \in \{1, 2\}$) to express the non-negativity of ReLUs, and $(0 \leq a_j)$ as described. These inequalities form the bound vectors l, u.

B. Proof Production

As a satisfiability problem, proofs of SAT are given directly by assignments that satisfy all constraints and are produced by many modern verifiers [46]. Proofs of UNSAT, however, are more involved and are typically represented as a *proof tree* [39]. In this tree, each node corresponds to a case split

over the two phases of a ReLU constraint, and its children represent the resulting subqueries. Each leaf of the proof tree contains a proof vector witnessing UNSAT for the corresponding subspace. This representation is based on a constructive variant of Farkas' lemma [39], [64], which applies to linear satisfiability problems defined by a matrix A together with lower and upper bounds l and u:

Theorem 1 (Farkas Lemma Variant (From [39])): Consider the query $A \cdot x = \bar{0}$ and $l \leq x \leq u$, for $A \in M_{m \times n}(\mathbb{R})$ and $l, x, u \in \mathbb{R}^n$. The query is UNSAT if and only if $\exists w \in \mathbb{R}^m$ such that for $w^\intercal \cdot A \cdot x \coloneqq \sum_{j=1}^{n} c_j \cdot x_j$, we have that $\sum_{c_j > 0} c_j \cdot u_j + \sum_{c_j < 0} c_j \cdot l_j < 0$ whereas $w^\intercal \cdot \bar{0} = 0$. Thus, w is a proof of UNSAT, and can be constructed during LP solving.

Finally, each node of the tree contains a sequence of *bound tightening lemmas*, which capture the derivation of variable bounds induced by piecewise-linear activation functions. Each lemma includes a *causing bound* used to infer a *derived bound*. These lemmas are proven by a proof vector, similar to the one in Theorem 1, for proving the causing bound, along with a proof rule representing the deduction of the derived bound from the causing one. The derived bound is applicable in the proof tree rooted in the node in which the lemma is learned. We conclude this section with an example of such a proof.

Example 4: Consider the query from Example 3, and the proof tree illustrated in Figure 2. The proof tree consists of three nodes (orange). Given the query Q, a single lemma (blue) is learned at the root, before splitting and deriving UNSAT (red) on both leaves. The lemma is proven by the vector $\begin{bmatrix}0 & -1 & 0 & 0 & 0\end{bmatrix}^\intercal$ representing the equation $-2i_2 - i_3 + b_2 = 0$, equivalent to $b_2 = 2i_2 + i_3$. Using the bounds, we can deduce that $l(b_2) = 2l(i_2) + l(i_3) = 0$. Given that b_2 is non-negative, we deduce that the constraint R_2 is always active, and that for its auxiliary variable a_2, $u(a_2) = 0$. The root has two children, representing splitting over the two phases of v_1, both leaves with an UNSAT vector and no lemmas. The right leaf, representing the case where v_1 is active, is proven with the vector $w = \begin{bmatrix}1 & -1 & -1 & -1 & 1\end{bmatrix}^\intercal$. Indeed, observe that:

$$\begin{aligned} w^\intercal \cdot A \cdot x = {} & i_1 - i_2 - b_1 \\ & - 2i_2 - i_3 + b_2 \\ & - f_2 + f_1 + o \\ & - f_1 + b_1 + a_1 \\ & + f_2 - b_2 - a_2 \\ = {} & i_1 - 3i_2 - i_3 + a_1 - a_2 + o \end{aligned}$$

The upper bound for this linear combination is then $u(i_1) - 3l(i_2) - l(i_3) + u(a_1) - l(a_2) + u(o) = 1 - 0 - 0 + 0 - 0 - 2 = -1 < 0$. Therefore, we deduce that w is a proof of UNSAT as in Theorem 1. UNSAT of the left leaf is proven similarly. Since all leaves are UNSAT, we deduce the UNSAT of Q.

C. The Alethe Proof Format

Alethe is a standard proof format for unsatisfiability in SMT solvers [9], [57]. Proofs in Alethe are expressed as sequences

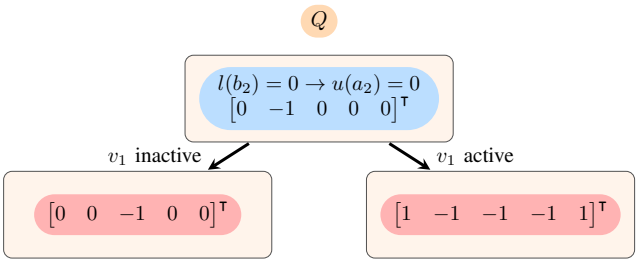

Fig. 2: A proof tree example, proving UNSAT for the query of Example 3

of *proof commands*, of which two are relevant here: (i) *assumptions*, which are first-order formulas importing assertions from the SMT problem; and (ii) *steps*, which correspond to propositional or first-order derivations. Each step describes the application of a well-defined proof *rule* that derives a clause from given *premises*, i.e., previous assumptions or steps.

As common in resolution proofs, an Alethe proof concludes with a step deriving the empty clause.

For example, our Alethe proofs extensively use the rule:

```
(step ite_i (cl φ1 φ3) :rule ite1 :premises(ite))
```

that deduces the clause φ_1, φ_3 from an ite statement ite φ_1 φ_2 φ_3, using the proof rule named ite1.

Alethe proofs can be translated (i.e., *reconstructed*) to proof terms in Isabelle/HOL [48], and a subset of their rules can also be translated to Rocq [28] and Lambdapi [20]. In addition, Alethe proofs can be checked by CARCARA [3], a widely used proof checker implemented in Rust. Although CARCARA is not formally verified, it is a mature, open-source tool that checks a set of fine-grained proof rules, uses arbitrary precision arithmetic and has a significantly smaller codebase compared to PICID. For these reasons, CARCARA provides a higher level of trust than most existing DNN verifiers, including PICID.

III. OVERVIEW OF PICID

Here, we outline the main novel components of PICID and their interactions, as illustrated in Fig. 3. In the subsequent sections we describe each of the components more thoroughly. Note that PICID builds upon, and extends, the existing Marabou tool; we use "Marabou" to refer to the existing baseline, and use $\mathcal{T}$-solver to refer to Marabou when used within the context of PICID.

Given an input query, PICID first performs a preprocessing pass, using Marabou modules, that transforms the given DNN and property into a set of linear and piecewise linear equations as described in Section II-A. It then initializes an instance of the novel *CdclCore* module, which is implemented within the $\mathcal{T}$-solver; and which is responsible for orchestrating its interaction with the SAT solver. To this end, CdclCore implements the $\mathcal{T}$-solver side of IPASIR-UP [30], [31], a general interface based on DPLL($\mathcal{T}$), which enables a modular integration of a SAT and $\mathcal{T}$- solvers. Adhering to the principles of DPLL($\mathcal{T}$), the SAT solver then begins to search for a satisfying Boolean assignment to its variables. During this search, the

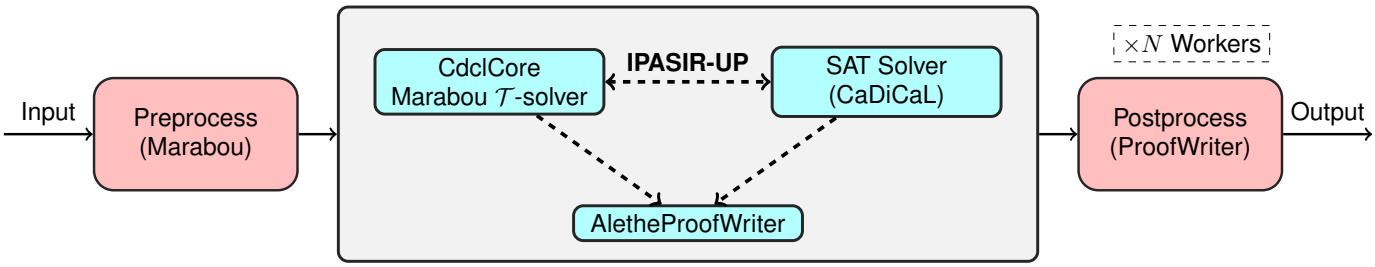

Fig. 3: An overview of PICID's architecture.

SAT solver either *notifies* the $\mathcal{T}$-solver on changes made in the Boolean level, or *calls back* the $\mathcal{T}$-solver for information. Upon callbacks and notifications from the SAT solver (e.g., $\mathcal{T}$-propagate), CdclCore invokes the corresponding routines in the $\mathcal{T}$-solver and relays the resulting information back to the SAT solver (e.g. assignment to Boolean variables). Proof generation is handled by the novel *AletheProofWriter* module, which collects proof information from both the SAT solver and the $\mathcal{T}$-solver and emits the corresponding proof steps.

If the query at hand is `SAT`, a satisfying assignment is returned and all proof information is deleted. Otherwise, if the query is `UNSAT`, the proof of unsatisfiability is readily available in the Alethe format, written in a dedicated file.

For handling complex queries, PICID is enhanced with a *split and conquer* (SNC) [69] parallelization mechanism that splits a query into several simpler subqueries, which are independently solved with concurrent workers. This mechanism is implemented within Marabou, and extended in PICID to support proof-producing CDCL($\mathcal{T}$) solving. When multiple workers are used in parallel, each worker is initialized with its own CdclCore, AletheProofWriter, $\mathcal{T}$- and SAT solvers. Notably, the workers do not perform additional preprocessing, ensuring a consistent Boolean abstraction across workers. During verification, all workers write to a shared proof file. If the query is `UNSAT` (i.e., all workers terminate with `UNSAT`), a designated worker additionally emits proof steps that derive the overall `UNSAT` result of the query from the `UNSAT` results of the subqueries. Otherwise, the proof file is deleted.

Currently, PICID can verify DNNs with the ReLU activation function. Extending it to any piecewise linear activation function is straightforward, using an appropriate Boolean abstraction and formulation for bound propagation rules in Alethe. Furthermore, Marabou's and consequently PICID's, preprocessing is not covered by proof production, and it requires a separate analysis. Producing such proofs was known to be a challenge in SMT solving as well [10]. We leave both directions for future work.

IV. CDCL($\mathcal{T}$) IMPLEMENTATION

A. DNN Verifiers as $\mathcal{T}$-Solvers

The DNN verification problem can be formulated as an SMT problem with a single theory solver of linear real arithmetic [44]. Separating the Boolean- and theory-specific reasoning in this way, as is generally done in SMT solving, is beneficial for multiple reasons: (i) It provides a clean logical separation of these two distinct levels of reasoning; (ii) It enables a modular integration of modern SAT solvers into DNN verifiers, allowing the latter to benefit from advances in SAT solving; and, (iii) It paves the way for future integration with additional theory solvers. Despite these facts, most modern DNN solvers do not follow this architecture. Instead, they either mix the two layers completely, or separate them entirely, so that very little information flows between the two.

Here, we advocate leveraging the experience gained by the SMT community in recent decades, by designing a DNN verifier according to the DPLL($\mathcal{T}$) architecture. Such a design has been attempted before [26], [27], [44], [49]; and we follow prior work and employ Boolean reasoning over the neurons of the DNN — using Booleans to encode the linear phases of activation functions. Consequently, unlike general-purpose SMT solvers, only a subset of the theory constraints is represented using Boolean variables, while the remaining constraints are handled internally by the $\mathcal{T}$-solver.

Building on this foundation, we take the next step and design PICID as a CDCL($\mathcal{T}$)-based DNN verifier. Conflict-Driven Clause Learning (CDCL) [13], [58], [59] extends DPLL by augmenting the formula with learned conflict clauses. Each clause corresponds to the negation of a partial assignment that led to unsatisfiability, and is therefore implied by the original formula. These clauses prevent the solver from revisiting subspaces of the search space that are guaranteed to contain no satisfying assignments. In SAT solving, CDCL is known to significantly prune the search tree and substantially improve scalability [13], [58], [59]. SMT solvers similarly rely on CDCL($\mathcal{T}$), the first-order extension of CDCL [12]. Prior work in DNN verification has proposed various techniques for detecting non-trivial conflict clauses [27], [74]. However, these attempts either involved repetitive invocations of an LP solver [27], or they were not implemented within a DPLL architecture [74]. To the best of our knowledge, our work is the first to focus on clause learning derived directly from `UNSAT` proofs. More importantly, our conflict clauses are learned alongside their corresponding proofs.

B. Proof-Driven Clause Learning

We next describe how `UNSAT` proofs can be used to derive conflict clauses. Conceptually, whenever Marabou determines that a subspace is `UNSAT`, the corresponding proof of unsatisfiability can be analyzed to extract a core set of input and decision constraints sufficient to imply `UNSAT`. Since the subset of decision constraints is abstracted by the SAT solver, it can be directly translated into a conflict clause. This approach has two key advantages: (i) It improves performance speed by learning non-trivial conflict clauses; and (ii) Each

conflict clause is accompanied by a minimized proof of its correctness. Below, we detail the adaptation of the proof-analysis algorithm of [41] to derive conflict clauses in PICID.

The goal of proof analysis is reducing the number of lemmas used to derive UNSAT, yielding more succinct proofs and, consequently, shorter conflict clauses. Conceptually, the analysis routine performs an on-the-fly backward analysis of the proof tree, starting from its leaves. When PICID derives UNSAT for a subquery, the corresponding proof is analyzed to identify which bounds are used in the proof, and which originate from bound-tightening lemmas. This allows detecting lemmas that were learned eagerly but are not actually used, and can therefore be safely removed. A minimization step then retains a minimal subset of such lemmas, resulting in its *UNSAT-core*. The procedure is applied recursively to the proofs of the retained lemmas and terminates once all required bounds are provided directly by the input constraints or by splitting decisions. Termination is ensured by assigning each lemma a unique chronological identifier. Then, the UNSAT-core analysis of each lemma is restricted to include only lemmas that were chronologically learned earlier. Adapting this process to derive conflict clauses is straightforward. Whenever a decision bound is encountered during the analysis, the negation of its corresponding (Boolean) decision variable is added to the clause. We formalize this adaptation in Algorithm 1 and omit the full description of the underlying minimization procedure from [41] for brevity. To avoid redundant computation, the clause derived for each lemma is cached upon its first analysis and reused in subsequent recursive calls.

Example 5: Recall the proof tree presented in Example 4. The right leaf is proven UNSAT using the proof vector $w = \begin{bmatrix}1 & -1 & -1 & -1 & 1\end{bmatrix}^\intercal$ that generates the equation $i_1 - 3i_2 - i_3 + a_1 - a_2 + o = 0$, which requires the bounds $u(i_1), l(i_2), l(i_3), u(a_1), l(a_2), u(o)$. From these bounds, only $u(a_1)$ is learned by the decision (v_1 is active). This means that in this case, the clause learned is the (trivial) clause $\neg v_1$, representing the fact the v_1 is inactive. Notably, the lemma learned in the root is not used for deriving UNSAT in both leaves, so Algorithm 1 will remove it from the proof.

Algorithm 1 *PDCL():* Construct proof-driven conflict clauses

Input: A DNN verification subquery $Q = \langle x, A, l, u, R\rangle$, a list of learned lemmas Lem, and a proof vector of UNSAT w
Output: A conflict clause $\mathcal{C}$.

// Let $\sum_{j=1}^{n} c_j \cdot x_j$ denote the linear combination $w^\intercal \cdot A \cdot x$
$Deps \leftarrow \emptyset$ ▷ An empty list of lemmas
$\mathcal{C} \leftarrow \emptyset$ ▷ The conflict clause
for $j \in [n]$ **do**
  if $c_j > 0$ and $u(x_j)$ is learned by $\ell_j^u \in Lem$ **then**
    $Deps \leftarrow Deps \cup \ell_j^u$
  else if $c_j < 0$ and $l(x_j)$ is learned by $\ell_j^l \in Lem$ **then**
    $Deps \leftarrow Deps \cup \ell_j^l$
  else if $c_j > 0$ and $u(x_j)$ is learned by a decision D **then**
    $\mathcal{C} \leftarrow \mathcal{C} \vee \neg D$
  else if $c_j < 0$ and $l(x_j)$ is learned by a decision D **then**
    $\mathcal{C} \leftarrow \mathcal{C} \vee \neg D$
  end if
end for
$Deps \leftarrow proofMin(Q, w, Deps, l, u)$ ▷ Minimize dependencies as in [41]
for $lem \in Deps$ **do** ▷ Repeat recursively
  $lem.includeInProof \leftarrow true$
  $l', u' \leftarrow$ bounds with ID at most $lem.getID()$
  $w' \leftarrow lem.getProof()$
  $\mathcal{C}$' $\leftarrow PDCL(\langle x, A', l', u', R\rangle, Lem, w')$
  $\mathcal{C} \leftarrow \mathcal{C} \vee \mathcal{C}$'
end for
return $\mathcal{C}$

C. IPASIR-UP Implementation

A key motivation behind our design is to allow a modular integration of SAT solvers into PICID. To maximize modularity, we use the IPASIR-UP interface [30], [31], which provides a structured way to connect $\mathcal{T}$-solvers with SAT solvers within a DPLL($\mathcal{T}$) framework. Though some SAT solvers support this interface (e.g., CaDiCaL [14]), to the best of our knowledge, no existing DNN verifier has implemented it thus far.

The IPASIR-UP interface includes several methods, some of which serve as notifications from the SAT solver to the $\mathcal{T}$-solver, informing it of key events during the SAT-level search process. These include notifications for assigned literals (`notify_assignment()`), backtracking to earlier decision levels (`notify_backtrack()`), and the introduction of new decision levels (`notify_new_decision_level()`). Separate methods serve as callbacks, allowing the SAT solver to query the $\mathcal{T}$-solver for relevant information. These include retrieving literals for propagation (`cb_propagate()`), selecting decision variables (`cb_decide()`), and providing explanatory clauses for previous propagations (`notify_add_reason_clause_lit()`) or conflicts (`notify_add_external_clause_lit()`).
Additionally, CaDiCaL allows integration of its internal proof logging with $\mathcal{T}$-solver proofs using two methods, `add_derived_clause` and `add_original_clause`. These methods inform that a Boolean resolution proof step is learned either by the SAT solver, or the $\mathcal{T}$-solver, respectively.

For PICID to be compatible with IPASIR-UP and support these methods, we maintain several data structures, including context-dependent ones (inspired by those used in Marabou [68] and CVC5 [8]). These data structures include a map representing the Boolean abstraction (`satSolverVarToPlc`), a list of deduced literals pending propagation to the SAT solver (`literalsToPropagate`), a context-dependent list of currently assigned literals (`assignedLiterals`), and another context-dependent list of satisfied clauses in the current context (`satisfiedClauses`). These structures ensure that whenever the SAT solver backtracks to an

earlier level in the search tree, the data within PICID reverts to the state it held at that level. Below, we describe the implementation of each method used in the basic module. The proof methods are described in Section V.

`void notify_assignment(const vector<int> &lits)`

This method processes a list of literals `lits` assigned by the SAT solver, notifying the $\mathcal{T}$-solver accordingly. The $\mathcal{T}$-solver iterates over each literal `lit` in `lits`, adding it to the internal list `assignedLiterals`. Subsequently, the corresponding piecewise-linear constraint (`plc`) is retrieved from `satSolverVarToPlc`. The phase of `plc` is fixed based on the sign of `lit`, which may lead to further tightening of $\mathcal{T}$-variable bounds. Finally, `satisfiedClauses` is updated accordingly to the new assignments.

`void notify_new_decision_level()`

This method is used to preserve the current state in context-dependent data structures. In particular, it stores `assigned Literals` and `satisfiedClauses` into their designated data structures, ensuring consistency across decision levels.

`void notify_backtrack(size_t new_level)`

This method facilitates backtracking to the specified decision level `new_level`. It restores the internal state of all context-dependent structures to align with the state corresponding to the given decision level. In particular, this includes reverting both `assignedLiterals` and `satisfiedClauses` to their respective states at `new_level`, and clearing the pending `literalsToPropagate`.

`bool cb_check_found_model(const vector<int> &model)`

This method verifies whether the complete boolean assignment `model`, provided by the SAT solver, satisfies all constraints encoded in the $\mathcal{T}$-solver. The $\mathcal{T}$-solver applies the functionality described in `notify_assignment` to process each previously unassigned literal in `model`. Then, the $\mathcal{T}$-solver is invoked to determine the satisfiability of the constraints imposed by `model`. If it identifies a satisfying assignment for all variables under the phase fixings defined by `model`, the method returns `true`. Conversely, if it concludes that no such assignment exists, the method returns `false`.

`int cb_decide()`

This method is invoked by the SAT solver to determine the next decision literal. In PICID, we integrate the VSIDS decision heuristic for SAT solvers [51] with the baseline heuristic used in Marabou. A score is computed for each literal based on the applied decision heuristics and the information stored in `satisfiedClauses`. Then, the literal with the highest score, representing a linear phase of an activation constraint, is returned as the decision literal.

`int cb_propagate()`

This method is invoked by the SAT solver to request additional literals for assignment that may have been deduced by the $\mathcal{T}$-solver. The $\mathcal{T}$-solver begins by checking whether the list of pending literals for propagation, `literalsToPropagate`, is empty. If `literalsToPropagate` is empty, the $\mathcal{T}$-solver is executed to deduce new phase fixings, which are then added to `literalsToPropagate` for subsequent propagation.

The method is repeatedly called to retrieve one literal for propagation at a time until no further literals remain. For each invocation, a single literal from `literalsToPropagate` is propagated. When `literalsToPropagate` becomes empty, the method signals this to the SAT solver by returning 0. Subsequent invocations of `cb_propagate()` will repeat this process, ensuring that all newly deduced literals are propagated until no further deductions are possible.

`int cb_add_reason_clause_lit(int propagated_lit)`

This method requests a reason clause from the $\mathcal{T}$-solver, providing an explanation for the previously propagated literal `propagated_lit`. The explanation provided to the SAT solver is constructed using Algorithm 1, applied to the $\mathcal{T}$-lemma that fixed the phase of the corresponding constraint. Similar to the process described in `cb_propagate`, the reason clause is constructed and propagated one literal at a time. The end of the clause is signaled by returning 0.

`bool cb_has_external_clause()` / `int cb_add_external_clause_lit()`

These two methods are used to handle conflict clauses. `cb_has_external_clause()` checks whether a conflict clause was deduced by the $\mathcal{T}$-solver during propagation, as discussed in Section IV-B. If a conflict clause is available, `cb_add_external_clause_lit()` propagates one literal from the conflict clause at a time. The end of the conflict clause is signaled by returning 0.

V. PROOF PRODUCTION IN ALETHE FORMAT

PICID produces proofs in the Alethe format [9], [57] for various reasons: (i) It is a standard proof format for SMT solving, used by existing tools [8]. As such, it naturally lends itself to expressing proofs that follow the DPLL($\mathcal{T}$) scheme, integrating proof steps of a SAT solver with those of a $\mathcal{T}$-solver; (ii) It is readily supported by an independent proof checker, enhancing reliability and scalability [3]; and (iii) Marabou proofs can be expressed naturally in this format.

The module responsible for proof generation is *AletheProofWriter*, which receives information from both the SAT solver and the $\mathcal{T}$-solver, and writes corresponding proof steps. The AletheProofWriter is implemented within the $\mathcal{T}$-solver, and it implements the extension of IPASIR-UP for proof production. It receives two types of notifications from the SAT solver and then uses the notified information and data structures of the $\mathcal{T}$-solver to write proofs steps in Alethe. The first, $add\_derived\_clause$, informs AletheProofWriter that a Boolean resolution proof step is learned by the SAT solver. These steps are given in the LRAT proof format, and can be directly translated into Boolean resolution steps in

Alethe. The second, *add_original_clause*, informs AletheProofWriter whenever conflict and reason clauses are learned by the $\mathcal{T}$-solver. AletheProofWriter converts the proofs of the given clause, and all $\mathcal{T}$-lemmas required for proving that clause, into Alethe format. In particular, proofs of $\mathcal{T}$-lemmas are written lazily [45], only after PICID's analysis identifies them as a part of the minimized proof.

A. Expressing Marabou Proofs in Alethe

We discuss here the details of converting Marabou proofs into the Alethe format. This includes formalizing four main components: (i) DNN verification problem in the SMT standard format [11], readable by Alethe; (ii) Proofs of UNSAT for each subspace of the search tree; (iii) Proofs of bound propagation based on the ReLU function, namely $\mathcal{T}$-lemmas; and (iv) Proof of soundness of case splitting, based on phases of the activation functions.

Since the fourth component is abstracted into Boolean predicates and case splitting is handled by the SAT solver, the correctness of case splitting in PICID proofs is established within the LRAT part of the proof. Consequently, we focus on translating the first three components.

Formalization in SMT. PICID proofs are checked against a formalization of the DNN verification problem in the SMTLIB standard format [11]. Given a query $\langle x, A, l, u, R\rangle$, we encode $\langle x, A, l, u\rangle$ as a set of assertions in quantifier-free linear real arithmetic (QF_LRA). Equations of the constraint matrix A are represented by assertions of the form:

(assert (= (+ (* c_j1 x_j1) $\cdots$ (* c_jk x_jk)) 0.0))

and bound are represented by inequalities of the form:

(assert (<= x_j u(x_j))) ; (assert (>= x_j l(x_j)))

For the constants $u(x_j), l(x_j)$. Following a standard formulation used in prior work [44], ReLU constraints of the form $f = \text{ReLU}(b)$ are encoded as:

(assert (ite (>= b 0.0) (= b f) (<= f 0.0)))

Proofs by the Farkas Lemma. Recall that UNSAT for each subspace of the search tree is proven by the Farkas lemma as in Theorem 1. Alethe, on the other hand, supports another variant of the Farkas lemma, captured in the linear arithmetic proof rule la_generic, formalized as follows:

(step i (cl $\varphi_1 \cdots \varphi_n$) :rule la_generic :args($a_1 \cdots a_n$))

where φ_i are the linear arithmetic literals of the derived clause, and args are the Farkas coefficients used to derive UNSAT of the negated clause. Notably, every literal in the clause is coupled with a coefficient. In Marabou, however, w describes only the coefficients of rows of A and not of l, u.

To translate a proof vector w into the Alethe format, we first compute the linear combination $\sum_{i=1}^{n} c_i \cdot x_i$. We then construct the corresponding clause by negating: (i) The rows A_j for which the associated coefficient $w_j \neq 0$; (ii) Upper-bound constraints $u_k - x_k \geq 0$ with positive coefficients c_k; and (iii) Lower-bound constraints $x_k - l_k \geq 0$ with negation of the negative coefficients c_k.

To ensure numerical stability, this procedure is implemented using an arbitrary-precision arithmetic library [35], in contrast to other components of the $\mathcal{T}$-solver. Below, we formalize this reduction.

Reduction of Farkas Vectors. In Alethe, the la_generic rule is checked by defining a sequence of atoms $\varphi_1, \ldots, \varphi_n$ together with corresponding coefficients. Then, the weighted linear combination is reduced to the contradiction $c > 0$ or $c \geq 0$ for some negative constant c [9].

To translate Marabou proof vectors as defined in Theorem 1 into the Alethe format, we instantiate atoms using rows of the constraint matrix A and bound expressions of the form $x_i - l_i \geq 0$ and $u_i - x_i \geq 0$ for lower and upper bounds, respectively. The remaining task is to compute the appropriate coefficients for each of these expressions. Formally, we prove:

Theorem 2 (Reduction of Marabou Proof Vectors to Alethe): Consider the query $A \cdot x = \bar{0}$ and $l \leq x \leq u$, for $A \in M_{m\times n}(\mathbb{R})$ and $l, x, u \in \mathbb{R}^n$. Let $\mathcal{L}$,$\mathcal{U}$ be a list of n expressions, whose j^{th} entry is $x_j - l_j, u_j - x_j$ respectively. If $\exists w \in \mathbb{R}^m$ such that for $w^\intercal \cdot A \cdot x := \sum_{j=1}^{n} c_j \cdot x_j$, we have that $\sum_{c_j>0} c_j \cdot u_j + \sum_{c_j<0} c_j \cdot l_j < 0$, then we can construct a proof vector $(w_1, w_2, w_3) \in \mathbb{R}^{m+2n}$ such that $w_1^\intercal \cdot A \cdot x + w_2^\intercal \cdot \mathcal{L} + w_3^\intercal \cdot \mathcal{U}$ is reduced to a negative constant.

Proof: Let $w_1 := w$, and consider the linear combination $w^\intercal \cdot A \cdot x := \sum_{j=1}^{n} c_j \cdot x_j$. For w_2, its j^{th} entry is $-c_j$ if and only if $c_j < 0$ and zero otherwise. For w_3, its j^{th} entry is c_j if and only if $c_j > 0$ and zero otherwise. Observe that:

$$w_1^\intercal \cdot A \cdot x + w_2^\intercal \cdot l + w_3^\intercal \cdot u =$$
$$\sum_{j=1}^{n} c_j \cdot x_j + \sum_{c_j<0} (-c_j) \cdot (x_j - l_j) + \sum_{c_j>0} c_j \cdot (u_j - x_j) =$$
$$\sum_{c_j<0} c_j \cdot x_j + \sum_{c_j<0} c_j \cdot (l_j - x_j) + \sum_{c_j>0} c_j \cdot x_j + \sum_{c_j>0} c_j \cdot (u_j - x_j) =$$
$$\sum_{c_j>0} c_j \cdot u_j + \sum_{c_j<0} c_j \cdot l_j$$

We assumed the last term is a negative constant. □

To improve proof compactness, when writing the proof step we use (w_1, w_2, w_3) after omitting all zero coefficients and their corresponding constraints. By definition, these are unnecessary for the proof.

Example 6: Consider $w = \begin{bmatrix}1 & -1 & -1 & -1 & 1\end{bmatrix}^\intercal$ illustrated in Figure 2, which generates the equation $i_1 - 3i_2 - i_3 + a_1 - a_2 + o = 0$ and deduce UNSAT. The corresponding Alethe proof step is as follows: Its clause consist of negation literals for all equations of A whose corresponding entry in w is non zero (i.e., all equations), the upper bounds $i_1 \leq 1$, $a_1 \leq 0$, $o \leq -2$ and the lower bounds $0 \leq i_2$, $0 \leq i_3$, $0 \leq a_2$. The corresponding arguments for the proof rule are (w_1, w_2, w_3) for $w_1 = w$, $w_2 = \begin{bmatrix}3 & 1 & 1\end{bmatrix}^\intercal$ and $w_3 = \begin{bmatrix}1 & 1 & 1\end{bmatrix}^\intercal$.

Proofs of $\mathcal{T}$-Lemmas. Recall that $\mathcal{T}$-lemmas describe the derivation of a *derived bound* from a previously established *causing bound*, together with a proof rule that justifies this derivation. Consequently, proving a $\mathcal{T}$-lemma consists of two parts: first, a proof of the causing bound, and second, Alethe proof steps that encode the corresponding Marabou proof rule. The first part is proven in Marabou using the Farkas lemma, and its translation to Alethe follows the reduction described above. The second part relies primarily on Boolean reasoning over ReLU phases and the bounds they induce. An example for such derivation appears in Figure 4.

Handling Holes. Before producing a candidate proof vector w, Marabou performs a sanity check based on the Farkas lemma to validate its correctness. If the proof vector passes this check, it is written as part of the proof; otherwise, Marabou deems the constructed proof vector invalid, and the corresponding case split is marked as *delegated* to external, numerically stable solvers. Those, in turn, could either correct the proof, or reveal errors made in the verification process. Prior work indicates that such delegation is rare [39], [41]. In the Alethe format, this situation is represented using the `hole` rule, indicating that the derivation of a given clause is left unproven. Such proof steps are of the following form:

$$(\texttt{step}\ \texttt{j}\ (\texttt{cl}\ \varphi_1 \cdots \varphi_n)\ \texttt{:rule}\ \texttt{hole})$$

Example. For completeness, an example for a PICID proof in the Alethe format, formalizing the proof of Example 4, appears in Figure 4. The proof begins with *assumption* commands representing the constraints of the verification query $Q = \langle x, A, l, u, R\rangle$. Lines 2–6 encode the equations of the matrix A, lines 9–19 specify all initial necessary bound constraints l and u, and lines 22–23 encode the ReLU constraints for neurons v_1 and v_2. The proof uses the `:named` tag to assign identifiers to terms, improving readability. In particular, we denote equations by e_j and ReLU constraints by R_j for some index j. The proof then introduces auxiliary steps, in lines 26-33, that derive equalities such as $u(a_j) = 0$ from $u(b_j) = 0$, and vice versa, which are both valid in the active phase of the corresponding ReLU. Then, lines 36–38 present a sequence of steps establishing a bound-tightening lemma. It begins with an application of the Farkas lemma (`la_generic`), showing that the negation of the causing bound leads to `UNSAT`. This is followed by resolution steps deriving both the causing bound and the derived bound. Next, in lines 41-51 the proof derives `UNSAT` for each leaf of the proof tree. As before, each derivation consists of an application of the Farkas lemma followed by resolution. Finally, the proof concludes by deriving the empty clause, combining the unsatisfiability of both leaves. Note that `lemma3` is derived, though it is not used as a premise of any other step. In practice, this bound-tightening lemma would be removed from the proof by Algorithm 1, and its steps will not appear in the Alethe file.

VI. PARALLELIZATION

Another key feature of PICID is the integration of a (SNC) [69] parallelization scheme, enabling concurrent solving of queries within the proof producing CDCL($\mathcal{T}$) framework. The core idea is to split complex queries into M simpler subqueries, based on common decision heuristics, and handle each of them independently. The subqueries are placed in a shared work queue, and a pool of N worker threads (typically $N \leq M$) repeatedly retrieves subqueries from this queue and solves them independently. This process continues until the queue is exhausted.

The global result is determined by aggregating the outcomes of all subqueries. If any worker identifies a satisfying assignment for its assigned subquery, the original query is immediately declared `SAT`. Conversely, the original query is `UNSAT` only after all subqueries teminated with `UNSAT`.

To support CDCL($\mathcal{T}$) solving and Alethe proof production within this setting, we instantiate a dedicated SAT solver instance per subquery. Each solver is initialized with the assumptions corresponding to the specific subquery it is responsible for. In parallel, each worker also creates a dedicated AletheProofWriter instance, which records the proof steps associated with that subquery. In `UNSAT` queries, upon the termination of all workers, a single worker constructs the final concluding proof steps for the original query. These final steps combine the concluding steps of all subquery proofs, into a complete proof of `UNSAT` for the original verification query.

VII. EVALUATION

We evaluated PICID along two axes: first, its scalability as a reliable verifier, measured by both verification time and `UNSAT`-proof checking time; and second, the correctness of the proofs, and their ability to detect flaws in the verification process.

As a baseline, we considered the standard proof-producing variant of Marabou extended with proof generation in Alethe.[1] To assess the overhead of producing Alethe proofs, we also evaluated Marabou's native proof-producing version. Additionally, we evaluated two configurations of PICID: with and without parallelization. This setup allows us to separately assess the impact of CDCL-based reasoning and parallel solving. All generated `UNSAT` proofs were checked by CARCARA using a single thread.

Finally, we evaluated our tool compared to NEURALSAT [26], a state-of-the-art proof-producing DNN verifier. NeuralSAT employs incomplete methods that significantly improve performance though are inherently more complex to prove, resulting in compact and less detailed proofs as we discuss in Section VIII. Due to technical limitations, we were able to successfully generate NeuralSAT proofs for only a single benchmark, on which we report. These proofs, written in an ad-hoc format, were validated using NeuralSAT's dedicated checker [25], which employs floating-point arithemtic.

[1] Another possible baseline is the proof-producing version of Marabou used in [23]; however, it is highly unoptimized and thus far less competitive.

```
; Constraint matrix
(assume e1 (!(= (+ i1 (- i2) (- b1)) 0.0):named e1))
(assume e2 (!(= (+ (* 2.0 i2) i3 (- b2)) 0.0):named e2))
(assume e3 (!(= (+ f2 (- f1) (- o)) 0.0):named e3))
(assume e4 (!(= (+ f1 (- b1) (- a1)) 0.0):named e4))
(assume e5 (!(= (+ f2 (- b2) (- a2)) 0.0):named e5))

; Bound constraints
(assume u_i1 (!(<= i1 1):named u_i1))
(assume l_i1 (!(>= i1 0):named l_i1))
(assume u_i2 (!(<= i2 1):named u_i2))
(assume l_i2 (!(>= i2 0):named l_i2))
(assume u_i3 (!(<= i3 1):named u_i3))
(assume l_i3 (!(>= i3 0):named l_i3))
(assume l_f1 (!(>= f1 0):named l_f1))
(assume l_f2 (!(>= f2 0):named l_f2))
(assume l_a1 (!(>= a1 0):named l_a1))
(assume l_a2 (!(>= a2 0):named l_a2))
(assume u_o (!(<= o -2):named u_o))

; Relu constraints
(assume r1 (!(ite (>= b1 0.0) (= b1 f1) (<= f1 0.0 )):named r1))
(assume r2 (!(ite (>= b2 0.0) (= b2 f2) (<= f2 0.0 )):named r2))

; Auxiliary steps
(step aux1 (cl (>= b1 0.0)(<= f1 0.0)):rule ite1 :premises(r1))
(step aux2 (cl (not (>= b1 0.0))(= b1 f1)):rule ite2 :premises(r1))
(step aux3 (cl (not (= b1 f1))(not e4)(<= a1 0.0)):rule la_generic :args(1 1 1))
(step aux4 (cl (not (>= b1 0.0))(<= a1 0.0)):rule resolution :premises(aux2 aux3 e4))

(step aux5 (cl (not (>= b2 0.0))(= b2 f2)):rule ite2 :premises(r2))
(step aux6 (cl (not (= b2 f2))(not e5)(<= a2 0.0)):rule la_generic :args(1 1 1))
(step aux7 (cl (not (>= b2 0.0))(<= a2 0.0)):rule resolution :premises(aux5 aux6 e5))

; Lemma derivation
(step lemma1 (cl (not e2)(not l_i2)(not l_i3)(>= b2 0.0)):rule la_generic :args(-1 -2 1 1))
(step lemma2 (cl (>= b2 0.0)):rule resolution :premises(lemma1 e2 l_i2 l_i3))
(step lemma3 (cl (<= a2 0.0)):rule resolution :premises(lemma2 aux7))

; Right leaf
(step right1 (cl (not e1)(not e2)(not e3)(not e4)(not e5)(not l_i2)(not l_i3)(not l_a2)
            (not u_i1)(not u_o)(not (<= a1 0.0))):rule la_generic
            :args(1 -1 -1 -1 1 3 1 1 1 1 1))
(step right2 (cl (not (<= a1 0.0))):rule resolution
            :premises(e1 e2 e3 e4 e5 l_i2 l_i3 l_a2 u_i1 u_o right1))
(step right3 (cl (not (>= b1 0.0))):rule resolution :premises(right2 aux4))

; Left leaf
(step left1 (cl (not e3)(not l_f2)(not u_o)(not (<= f1 0.0))):rule la_generic :args(-1 1 1 1))
(step left2 (cl (not (<= f1 0.0))):rule resolution :premises(left1 e3 l_f2 u_o))
(step left3 (cl (>= b1 0.0)):rule resolution :premises(aux1 left2))

; Finalization
(step final (cl):rule resolution :premises(right3 left3))
```

Fig. 4: An example for a PICID proof in Alethe

Note that PICID could also be compared to proof-producing SMT solvers such as CVC5 [8]. However, prior work has shown that such solvers do not scale beyond small DNNs with only dozens of neurons [44]. Our preliminary experiments confirm this observation (e.g., CVC5 did not solve any ACASXU query within two hours), and so we omit these results.

All experiments were conducted on CPU machines running Linux Debian 12. We allocated a memory limit of 32GB for verification and 64GB for proof checking. We imposed a verification timeout of 90 minutes for ACASXU and MNIST_FC, as they express queries over larger networks, and 30 minutes for CERSYVE and SAFENLP. Parallelization experiments were conducted using 16 cores.

We evaluate our approach on four benchmark suites used in VNN-COMP [16], [46]: ACASXU [43], comprising DNNs with 300 neurons; CERSYVE [72], with networks ranging from 64 to 198 neurons; SAFENLP [17], [18], consisting of DNNs with 128 neurons; and MNIST_FC [6], which includes DNNs with 512–1024 neurons. These benchmarks are widely used in the DNN verification literature and were selected to cover a diverse range of network size and application domains.

For CERSYVE, we evaluate all queries appearing in the benchmark suite over two commits of VNN-COMP benchmarks [65]. For MNIST_FC, the original benchmark suite comprises of queries over three fully connected network architectures with 2×256, 4×256, and 6×256 hidden layers. Here, we consider only the queries corresponding to the two smaller architectures. Moreover, each MNIST_FC property is expressed as a disjunction of nine constraints. As disjunction is not yet natively supported in PICID, we preprocess each query

by breaking it down into nine subqueries, one per disjunct.

Table I summarizes the results for all benchmarks and configurations. Additional graphs detailing our results appears in the extended version of this paper [40]. Below, we analyze our results with respect to the two evaluation questions. Note that some queries were deduced UNSAT during preprocessing, across all methods, and thus their proofs are not generated.

Performance Speed. Overall, PICID significantly outperforms Marabou in most cases, both in terms of the number of solved queries and in average verification and proof-checking times. In particular, PICID solved queries $2.3\times$-$5\times$ faster for UNSAT queries, and demonstrates mixed results of $0.66\times$-$2.9\times$ improvement factor for SAT queries on average. When considering SNC, performance is further improved, leading to an increased number of solved queries and reduced average verification times by $4\times$-$10.2\times$ for UNSAT queries and by $2.3\times$-$7.8\times$ for SAT queries on average. However, we observe that proof checking times are higher for PICID with SNC. We hypothesize this is due to the fact that worker threads contribute independent proof fragments, which are subsequently combined into a single proof. Therefore, enabling workers to share clauses and their proofs may lead to a decrease in both proof size and checking speed. We leave this to future work.

As expected, NeuralSAT outperforms PICID both in the number of solved queries in their average running time. However, PICID successfully produced proofs for a greater number of UNSAT instances, and these proofs were checked in a significantly shorter time. This comparison emphasizes the trade-off between scalability of DNN verifiers, and their ability to produce detailed proofs.

Comparing the Alethe-producing baseline to the native proof-producing Marabou, we observe that producing Alethe proofs incurs an overheads that increases verification time by a factor of $1.18\times$-$1.6\times$ for UNSAT queries and by $1.02\times$-$2\times$ for SAT queries on average. This overhead is expected, due to the involved file writing and the computation of the vector reduction as in Theorem 2. However, this overhead is significantly diminished by the performance improvements of PICID.

Proof Correctness. Overall, on three out of four benchmarks, all proofs whose checking terminated within the allotted time and memory constraints were found correct by CARCARA, besides a single unstable query of ACASXU, as previously known [39], [41]. The CERSYVE benchmark, however, exhibits cases where PICID and Marabou are numerically unstable. This is reflected in three scenarios: (i) In several cases, both tools detect this instability, resulting in the generation of proof holes. These proofs are reported as *holey*, meaning all other proof steps were correctly checked; (ii) Two queries that are solved as UNSAT only by PICID, yield invalid proofs, detected by CARCARA; and (iii) In one instance, Marabou reports SAT while PICID incorrectly reports UNSAT [46], accompanied with a holey proof. The holes of the proof correspond to the potential subspaces that were incorrectly solved, possibly due to numerical instability, thus narrowing the possible states that lead to error.

Overall, PICID was able to generate valid proofs for more than 99% of the cases. In all other cases, the proof-checking mechanism successfully detected the flaws, either by failing to construct correct proofs, or by generating proof holes that pinpoint the potentially erroneous steps.

VIII. CONCLUSION

We present PICID, a proof-producing DNN verifier that implements a parallel CDCL($\mathcal{T}$) architecture in which conflict clauses are derived from UNSAT proofs. PICID is the first DNN verifier to generate proofs in a standard SMT format. Notably, PICID does not scale as state-of-the-art verifiers, since it does not employ symbolic, incomplete verification methods, which are intrinsically complex to prove. Nonetheless, PICID pushes forward both scalability and standardization of proof-producing DNN verification.

Related Work. Several DNN verifiers employ CDCL($\mathcal{T}$)-like verification schemes [26], [27], [49], [74], leveraging the analogy between ReLU activation phases and Boolean assignments. Notably, these verifiers either rely on naive clause-learning or first detect UNSAT cases and subsequently reinvoke solving procedures to minimize clause size [19], [74]. In contrast, our approach directly exploits information from UNSAT proofs to derive non-trivial conflict clauses.

One verifier of *Binarized* Neural Networks (BNNs) [71], uses a SAT-based approach to verify, produce and reliably check proofs. This work is tailored to BNNs while our work focuses on verifying a broader class of DNNs.

Most closely related to our work is the NeuralSAT DNN verifier [26], which employs a CDCL($\mathcal{T}$) architecture and is able to produce UNSAT proofs [25]. However, NeuralSAT proofs are not directly comparable to ours: they are checked through repeated invocations of an MILP solver, and require additional elaboration before they can be translated into a standard proof format. Furthermore, the MILP solver involved in the checking process uses floating-point arithmetic, and thus it prone to the same reliability issues as NeuralSAT.

Other approaches to reliable DNN verification are presented in [2], [60], [62]. The work in [60] seeks to prove the correctness of abstract-interpretation-based DNN verifiers, formalized in a dedicated domain-specific language. In contrast, PICID produces SMT-like proofs for individual verification queries, while making fewer assumptions regarding the internal workings of the verifier. Consequently, the two approaches are not directly comparable. The work in [62] presents a formally-verified algorithm that verifies the Lipschitz-continuity-based robustness of DNNs. Although it provides a reliable verification procedure, it targets a single class of properties and relies on an incomplete method. In contrast, PICID is complete and supports verification of any QF_LRA property. The Rocq-NN-Roll prover, recently presented in [2], is a DNN verifier implemented and formally verified in Rocq. It implements reliable verification procedures as well, though it scales up to DNNs containing only a few neurons.

TABLE I: Evaluation results per benchmark. We report the number of solved queries, average verification and checking times, and proof validity (VALID/HOLEY/INVALID/OUT OF RESOURCES/NOT GENERATED). Best performing results across comparable measurements are marked in **bold**. Avg. Shared represent averaging results for queries solved by all methods.

Method	Solved (#)		Avg. Verification (s)		Avg. Shared Verification (s)		Avg. Check (s)	Avg. Shared Check (s)	Validity (V/H/I/O/N)
	SAT	UNSAT	SAT	UNSAT	SAT	UNSAT			
ACAS Xu (180 queries)									
Marabou	40	87	327.71	591.71	327.71	365.89	N/A	N/A	N/A
Marabou + Alethe	40	82	467.06	522.05	467.06	522.06	395.45	395.45	82/0/0/0/0
PICID	42	95	475.65	631.34	241.38	226.46	503.03	**113.68**	94/1/0/0/0
PICID with SNC	**45**	110	203.59	618.17	**59.43**	128.34	1014.09	287.5	98/1/0/11/0
NeuralSAT	**45**	**135**	123.75	76.05	101.57	**48.70**	2815.53	730.58	58/0/0/22/55
CERSYVE (20 queries)									
Marabou	**10**	8	15.04	226.37	15.04	104.39	N/A	N/A	N/A
Marabou + Alethe	**10**	7	30.32	173.83	30.32	173.83	103.25	103.25	5/2/0/0/0
PICID	**10**	8	10.56	128.33	10.56	64	45.37	**27.02**	5/1/2/0/0
PICID with SNC	**10**	**10**	17.96	157.67	**3.86**	**29.68**	71.04	27.28	6/2/2/0/0
SAFENLP (1080 queries)									
Marabou	577	209	2.89	117.6	2.89	60	N/A	N/A	N/A
Marabou + Alethe	578	202	5.89	93.83	2.95	93.83	61.82	61.82	156/0/0/0/46
PICID	620	246	19	165.11	1.36	18.7	67.20	**10.48**	200/0/0/0/46
PICID with SNC	**632**	**263**	8.06	123.59	**1.3**	**9.18**	243.67	14.66	216/0/0/1/46
MNIST_FC (540 queries)									
Marabou	35	165	752.99	322.55	487.97	290.39	N/A	N/A	N/A
Marabou + Alethe	36	165	957.06	367.99	559.33	342.58	9.26	8.35	157/0/0/0/8
PICID	25	209	844.81	367.32	844.81	75.19	18.3	**2.90**	201/0/0/0/8
PICID with SNC	**54**	**215**	852.58	520.29	**149.55**	**50.39**	263.19	7.88	207/0/0/0/8

Finally, a variety of parallelization techniques have been explored in SMT solving and DNN verification [67], [69]. To the best of our knowledge, our approach is the first technique for DNN verification that inherently produces proofs.

Future Work. We plan to extend PICID along several directions. First, we plan to extend proof production to incomplete verification methods, enabling support for a broader range of optimizations and non-linear activation functions. Second, we aim to support additional piecewise-linear activation functions, and produce proofs for Marabou's preprocessing. Third, we aim to further reduce PICID's proof size by allowing workers to share learned clauses together with their associated proofs, reducing redundant derivations of similar lemmas. We believe that combining these directions will yield a faster and more versatile proof-producing verifier, capable of scaling to larger DNNs than is currently possible.

Acknowledgments. The work of Isac, Refaeli and Katz was supported by the European Union (ERC, VeriDeL, 101112713). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them. The work of Wu is supported by a gift from VMware University Research Fund. The work of Barrett was supported in part by the National Science Foundation (grant number 2211505), and the Stanford Center for AI Safety.

The authors wish to thank Armin Biere, Katalin Fazekas, Mathias Fleury, and Mathias Priener for the guidance on using IPASIR-UP, and to Bruno Andreotti and Haniel Barbosa for their guidance on generating Alethe proofs and using Carcara.

REFERENCES

[1] M. Akintunde, A. Kevorchian, A. Lomuscio, and E. Pirovano. Verification of RNN-Based Neural Agent-Environment Systems. In *Proc. 33rd AAAI Conf. on Artificial Intelligence (AAAI)*, pages 197–210, 2019.

[2] A. Aleksandrov, M. Jackisch, and K. Völlinger. The Rocq-NN-Roll Prover: Soundly Verifying Hyperproperties of Neural Networks in Rocq. In *Proc. 38th Int. Conf. on Computer Aided Verification (CAV)*, pages 480–503, 2026.

[3] B. Andreotti, H. Lachnitt, and H. Barbosa. Carcara: An Efficient Proof Checker and Elaborator for SMT Proofs in the Alethe Format. In *Proc. 29th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 367–386, 2023.

[4] G. Avni, R. Bloem, K. Chatterjee, T. Henzinger, B. Konighofer, and S. Pranger. Run-Time Optimization for Learned Controllers through Quantitative Games. In *Proc. 31st Int. Conf. on Computer Aided Verification (CAV)*, pages 630–649, 2019.

[5] S. Bak. nnenum: Verification of Relu Neural Networks with Optimized Abstraction Refinement. In *Proc. 13th NASA Formal Methods Symposium (NFM)*, pages 19–36, 2021.

[6] S. Bak, C. Liu, and T. Johnson. The Second International Verification of Neural Networks Competition (VNN-COMP 2021): Summary and Results, 2021. Technical Report. http://arxiv.org/abs/2109.00498.

[7] T. Baluta, S. Shen, S. Shinde, K. Meel, and P. Saxena. Quantitative Verification of Neural Networks And its Security Applications. In *Proc. 26th ACM Conf. on Computer and Communication Security (CCS)*, 2019.

[8] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar. CVC5: A Versatile and Industrial-Strength SMT Solver. In *Proc. 28th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 415–442, 2022.

[9] H. Barbosa, M. Fleury, P. Fontaine, and H.-J. Schurr. The Alethe Proof Format. https://verit.loria.fr/documentation/alethe-spec.pdf.

[10] C. Barrett, L. de Moura, and P. Fontaine. Proofs in Satisfiability Modulo Theories. *All about Proofs, Proofs for All*, 55(1):23–44, 2015.

[11] C. Barrett, P. Fontaine, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[12] C. Barrett and C. Tinelli. Satisfiability Modulo Theories. In E. Clarke, T. Henzinger, H. Veith, and R. Bloem, editors, *Handbook of Model Checking*, pages 305–343. Springer International Publishing, 2018.

[13] R. Bayardo Jr and R. Schrag. Using CSP Look-Back Techniques to Solve Real-World SAT Instances. In *Proc. 14th Nat. Conf. on Artificial Intelligence (AAAI)*, pages 203–208, 1997.

[14] A. Biere, T. Faller, K. Fazekas, M. Fleury, N. Froleyks, and F. Pollitt. CaDiCaL 2.0. In *Proc. 36th Int. Conf. on Computer Aided Verification (CAV)*, pages 133–152, 2024.

[15] C. Brix, S. Bak, T. T. Johnson, and H. Wu. The fifth International Verification of Neural Networks Competition (VNN-COMP 2024): Summary and Results, 2024. Technical Report. https://arxiv.org/abs/2412.19985.

[16] C. Brix, M. Müller, S. Bak, T. Johnson, and C. Liu. First Three Years of the International Verification of Neural Networks Competition (VNN-COMP). *Int. Journal on Software Tools for Technology Transfer*, pages 1–11, 2023.

[17] M. Casadio, L. Arnaboldi, M. Daggitt, O. Isac, T. Dinkar, D. Kienitz, V. Rieser, and E. Komendantskaya. ANTONIO: Towards a Systematic Method of Generating NLP Benchmarks for Verification. In *Proc. 6th Workshop on Formal Methods for ML-Enabled Autonomous Systems (FoMLAS)*, pages 59–70, 2023.

[18] M. Casadio, T. Dinkar, E. Komendantskaya, L. Arnaboldi, M. L. Daggitt, O. Isac, G. Katz, V. Rieser, and O. Lemon. NLP Verification: Towards a General Methodology for Certifying Robustness. *European Journal of Applied Mathematics*, page 1–58, 2025.

[19] J. Chinneck and E. Dravnieks. Locating Minimal Infeasible Constraint Sets in Linear Programs. *ORSA Journal on Computing*, 3(2):157–168, 1991.

[20] A. Coltellacci, G. Dowek, and S. Merz. Reconstruction of SMT proofs with Lambdapi. In *Proc. 22nd Int. Workshop on Satisfiability Modulo Theories (SMT)*, pages 13–23, 2024.

[21] L. Cruz-Filipe, M. J. Heule, W. A. Hunt Jr, M. Kaufmann, and P. Schneider-Kamp. Efficient Certified RAT Verification. In *Proc. 26th Int. Conf. on Automated Deduction (CADE)*, pages 220–236, 2017.

[22] G. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.

[23] R. Desmartin, O. Isac, E. Komendantskaya, K. Stark, G. Passmore, and G. Katz. A Certified Proof Checker for Deep Neural Network Verification in Imandra. In *Proc. 16th Int. Conf. on Interactive Theorem Proving (ITP)*, pages 1–21, 2025.

[24] R. Desmartin, O. Isac, G. Passmore, K. Stark, E. Komendantskaya, and G. Katz. Towards a Certified Proof Checker for Deep Neural Network Verification. In *Proc. 33rd Int. Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR)*, pages 198–209, 2023.

[25] H. Duong and T. Nguyen. Generating and Checking DNN Verification Proofs. In *Proc. 39th Annual Conf. on Neural Information Processing Systems (NeurIPS)*, 2025.

[26] H. Duong, T. Nguyen, and M. Dwyer. NeuralSAT: A High-Performance Verification Tool for Deep Neural Networks, 2025. Proc. 37th Int. Conf. on Computer Aided Verification (CAV).

[27] R. Ehlers. Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks. In *Proc. 15th Int. Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 269–286, 2017.

[28] B. Ekici, A. Mebsout, C. Tinelli, C. Keller, G. Katz, A. Reynolds, and C. Barrett. SMTCoq: A Plug-in for Integrating SMT Solvers into Coq. In *Proc. 29th Int. Conf. Computer Aided Verification (CAV)*, pages 126–133, 2017.

[29] Y. Elboher, R. Elsaleh, O. Isac, M. Ducoffe, A. Galametz, G. Povéda, R. Boumazouza, N. Cohen, and G. Katz. Robustness Assessment of a Runway Object Classifier for Safe Aircraft Taxiing. In *Proc. 43rd Int. Digital Avionics Systems Conf. (DASC)*, 2024.

[30] K. Fazekas, A. Niemetz, M. Preiner, M. Kirchweger, S. Szeider, and A. Biere. IPASIR-UP: User Propagators for CDCL. In *Proc. 26th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT)*, pages 1–13, 2023.

[31] K. Fazekas, A. Niemetz, M. Preiner, M. Kirchweger, S. Szeider, and A. Biere. Satisfiability Modulo User Propagators. *Journal of Artificial Intelligence Research*, 81:989–1017, 2024.

[32] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL (T): Fast Decision Procedures. In *Proc. 16th Int. Conf. on Computer Aided Verification (CAV)*, pages 175–188, 2004.

[33] T. Gehr, M. Mirman, D. Drachsler-Cohen, E. Tsankov, S. Chaudhuri, and M. Vechev. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *Proc. 39th IEEE Symposium on Security and Privacy (S&P)*, 2018.

[34] M. Giacobbe, T. A. Henzinger, and M. Lechner. How Many Bits Does it Take to Quantize Your Neural Network? In *Proc. 26th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 79–97, 2020.

[35] GMP (The GNU Multiple Precision Arithmetic Library). https://gmplib.org.

[36] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

[37] M. J. Heule and A. Biere. Proofs for Satisfiability Problems. *All about Proofs, Proofs for All*, 55(1):1–22, 2015.

[38] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu. Safety Verification of Deep Neural Networks. In *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*, pages 3–29, 2017.

[39] O. Isac, C. Barrett, M. Zhang, and G. Katz. Neural Network Verification with Proof Production. In *Proc. 22nd Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 38–48, 2022.

[40] O. Isac, I. Refaeli, H. Wu, C. Barrett, and G. Katz. PICID: Proof-Driven Clause Learning in Neural Network Verification, 2026. Technical Report. https://arxiv.org/abs/2503.12083.

[41] O. Isac, I. Refaeli, H. Wu, C. Barrett, and G. Katz. Proof Minimization in Neural Network Verification. In *Proc. 27th Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 99–124, 2026.

[42] K. Jia and M. Rinard. Exploiting Verified Neural Networks via Floating Point Numerical Error. In *Proc. 28th Int. Static Analysis Symposium (SAS)*, pages 191–205, 2021.

[43] K. Julian, M. Kochenderfer, and M. Owen. Deep Neural Network Compression for Aircraft Collision Avoidance Systems. *Journal of Guidance, Control, and Dynamics*, 42(3):598–608, 2019.

[44] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Reluplex: a Calculus for Reasoning about Deep Neural Networks. *Formal Methods in System Design (FMSD)*, 2021.

[45] G. Katz, C. Barrett, C. Tinelli, A. Reynolds, and L. Hadarean. Lazy Proofs for DPLL(T)-Based SMT Solvers. In *Proc. 16th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 93–100, 2016.

[46] K. Kaulen, T. Ladner, S. Bak, C. Brix, H. Duong, T. Flinkow, T. T. Johnson, L. Koller, E. Manino, T. H. Nguyen, and H. Wu. The 6th International Verification of Neural Networks Competition (VNN-COMP 2025): Summary and Results, 2025. Technical Report. http://arxiv.org/abs/2512.19007.

[47] I. Khmelnitsky, D. Neider, R. Roy, X. Xie, B. Barbot, B. Bollig, A. Finkel, S. Haddad, M. Leucker, and L. Ye. Analysis of Recurrent Neural Networks via Property-Directed Verification of Surrogate Models. *Int. Journal on Software Tools for Technology Transfer*, 25(3):341–354, 2023.

[48] H. Lachnitt, M. Fleury, H. Barbosa, J. Jakpor, B. Andreotti, A. Reynolds, H.-J. Schurr, C. Barrett, and C. Tinelli. Improving the SMT Proof Reconstruction Pipeline in Isabelle/HOl. In *Proc. 16th Int. Conf. on Interactive Theorem Proving (ITP)*, pages 26–1, 2025.

[49] Z. Liu, P. Yang, L. Zhang, and X. Huang. DeepCDCL: A CDCL-based Neural Network Verification Framework. In *Proc. 18th Int. Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 343–355, 2024.

[50] Z. Lyu, C.-Y. Ko, Z. Kong, N. Wong, D. Lin, and L. Daniel. Fastened Crown: Tightened Neural Network Robustness Certificates. In *Proc. 34th AAAI Conf. on Artificial Intelligence (AAAI)*, pages 5037–5044, 2020.

[51] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proc. 38th Annual Design Automation Conf. (DAC)*, pages 530–535, 2001.

[52] F. Pollitt, M. Fleury, and A. Biere. Faster LRAT Checking than Solving with CaDiCaL. In *Proc. 26th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT)*, pages 21–1, 2023.

[53] L. Pulina and A. Tacchella. An Abstraction-Refinement Approach to Verification of Artificial Neural Networks. In *Proc. 22nd Int. Conf. on Computer Aided Verification (CAV)*, pages 243–257, 2010.

[54] I. Refaeli, S. M., I. Buchnik, A. Zada, G. Amir, E. Mandelbaum, F. Z., and G. Katz. veriFIRE: An Industrial Case Study in Verifying Consistency Properties for a DNN-Based Wildfire Detection System. In *Proc. 9th Int. Symposium on AI Verification (SAIV)*, 2026.

[55] S. Sankaranarayanan, S. Dutta, and S. Mover. Reaching Out Towards Fully Verified Autonomous Systems. In *Proc. 13th Int. Conf. on Reachability Problems (RP)*, pages 22–32, 2019.

[56] C. Schilling, M. Forets, and S. Guadalupe. Verification of Neural-Network Control Systems by Integrating Taylor Models and Zonotopes. In *Proc. 36th AAAI Conf. on Artificial Intelligence (AAAI)*, pages 8169–8177, 2022.

[57] H.-J. Schurr, M. Fleury, H. Barbosa, and P. Fontaine. Alethe: Towards a Generic SMT Proof Format. In *Proc. 7th Workshop on Proof eXchange for Theorem Proving (PxTP)*, pages 49–54, 2021.

[58] J. Silva and K. Sakallah. GRASP-A New Search Algorithm for Satisfiability. In *Proc. 15th IEEE/ACM Int. Conf. on Computer Aided Design (ICCAD)*, pages 220–227, 1996.

[59] J. M. Silva and K. A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[60] A. Singh, Y. C. Sarita, C. Mendis, and G. Singh. Automated Verification of Soundness of DNN Certifiers. *Proc. ACM on Programming Languages*, 9, 2025.

[61] G. Singh, T. Gehr, M. Puschel, and M. Vechev. An Abstract Domain for Certifying Neural Networks. In *Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2019.

[62] J. Tobler, H. T. Syeda, and T. Murray. A Formally Verified Robustness Certifier for Neural Networks. In *Proc. 37th Int. Conf. Computer Aided Verification (CAV)*, pages 327–348, 2025.

[63] H.-D. Tran, S. Bak, W. Xiang, and T. Johnson. Verification of Deep Convolutional Neural Networks Using ImageStars. In *Proc. 32nd Int. Conf. on Computer Aided Verification (CAV)*, pages 18–42, 2020.

[64] R. Vanderbei. Linear Programming: Foundations and Extensions. *Journal of the Operational Research Society*, 1998.

[65] The VNNCOMP 2025 Benchmarks Repository, 2025. https://github.com/VNN-COMP/vnncomp2025/benchmarks.

[66] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Formal Security Analysis of Neural Networks using Symbolic Intervals. In *Proc. 27th USENIX Security Symposium*, pages 1599–1614, 2018.

[67] S. Wang, H. Zhang, K. Xu, X. Lin, S. Jana, C.-J. Hsieh, and Z. Kolter. Beta-Crown: Efficient Bound Propagation with Per-Neuron Split Constraints for Neural Network Robustness Verification. In *Proc. 35th Annual Conf. on Neural Information Processing Systems (NeurIPS)*, pages 29909–29921, 2021.

[68] H. Wu, O. Isac, A. Zeljić, T. Tagomori, M. Daggitt, W. Kokke, I. Refaeli, G. Amir, K. Julian, S. Bassan, P. Huang, O. Lahav, M. Wu, M. Zhang, E. Komendantskaya, G. Katz, and C. Barrett. Marabou 2.0: A Versatile Formal Analyzer of Neural Networks. In *Proc. 36th Int. Conf. on Computer Aided Verification (CAV)*, 2024.

[69] H. Wu, A. Ozdemir, A. Zeljić, K. Julian, A. Irfan, D. Gopinath, S. Fouladi, G. Katz, C. Pasareanu, and C. Barrett. Parallelization Techniques for Verifying Neural Networks. In *Proc. 20th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 128–137, 2020.

[70] X. Xie, K. Kersting, and D. Neider. Neuro-Symbolic Verification of Deep Neural Networks. In *Proc. 31st Int. Joint Conf. on Artificial Intelligence (IJCAI)*, 2022.

[71] J. Yang, Y. K. Tan, M. Soos, M. O. Myreen, and K. S. Meel. Efficient Certified Reasoning for Binarized Neural Networks. In *Proc. 28th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT)*, 2025.

[72] Y. Yang, H. Hu, T. Wei, S. E. Li, and C. Liu. Scalable Synthesis of Formally Verified Neural Value Function for Hamilton-Jacobi Reachability Analysis. *Journal of Artificial Intelligence Research*, 83, 2025.

[73] H. Zhang, M. Shinn, A. Gupta, A. Gurfinkel, N. Le, and N. Narodytska. Verification of Recurrent Neural Networks for Cognitive Tasks via Reachability Analysis. In *Proc. 24th European Conf. on Artificial Intelligence (ECAI)*, pages 1690–1697, 2020.

[74] D. Zhou, C. Brix, G. A. Hanasusanto, and H. Zhang. Scalable Neural Network Verification with Branch-and-Bound Inferred Cutting Planes. In *Proc. 38th Annual Conf. on Neural Information Processing Systems (NeurIPS)*, pages 29324–29353, 2024.

[75] D. Zombori, B. Bánhelyi, T. Csendes, I. Megyeri, and M. Jelasity. Fooling a Complete Neural Network Verifier. In *Proc. 9th Int. Conf. on Learning Representations (ICLR)*, 2021.