Incremental Neural Network Verification via Learned Conflicts

Raya Elsaleh*, Liam Davis†, Haoze Wu†, Guy Katz*
* Hebrew University of Jerusalem, Jerusalem, Israel
{raya.elsaleh,g.katz}@mail.huji.ac.il
† Amherst College, Amherst, USA
{ljdavis27,hwu}@amherst.edu

***Abstract*—Neural network verification is often used as a core component within larger analysis procedures, which generate sequences of closely related verification queries over the same network. In existing neural network verifiers, each query is typically solved independently, and information learned during previous runs is discarded, leading to repeated exploration of the same infeasible regions of the search space. In this work, we aim to expedite verification by reducing this redundancy. We propose an incremental verification technique that reuses learned conflicts across related verification queries. The technique can be added on top of any branch-and-bound-based neural network verifier. During verification, the verifier records conflicts corresponding to learned infeasible combinations of activation phases, and retains them across runs. We formalize a refinement relation between verification queries and show that conflicts learned for a query remain valid under refinement, enabling sound conflict inheritance. Inherited conflicts are handled using a SAT solver to perform consistency checks and propagation, allowing infeasible subproblems to be detected and pruned early during search. We implement the proposed technique in the Marabou verifier and evaluate it on three verification tasks: local robustness radius determination, verification with input splitting, and minimal sufficient feature set extraction. Our experiments show that incremental conflict reuse reduces verification effort and yields speedups of up to 1.9× over a non-incremental baseline.**

I. Introduction

Deep neural networks (DNNs) are increasingly deployed in safety-critical applications [3], including autonomous driving [10], [32], medical diagnosis [21], and aerospace systems [19], [26]. Their strong empirical performance has driven major advances in tasks such as control and image recognition. Despite this success, neural networks remain largely opaque and difficult to reason about, raising serious concerns regarding their reliability and safety in critical settings.

To address this issue, numerous approaches have been developed for neural network verification, with the goal of providing rigorous guarantees about network behavior. For networks with piecewise-linear activation functions, such as ReLUs, the verification problem is NP-complete [26], [33], which hinders scalability; but as neural networks are adopted across increasingly diverse and high-stakes domains, the demand for scalable verification techniques continues to grow. Improving the efficiency of neural network verification is therefore imperative.

Examining how DNN verification is used in practice, we observe that it is often not performed as a single, isolated query, but rather invoked repeatedly within larger analysis procedures. For example, in formal explainability, verification queries are issued repeatedly to reason about the contribution of different input features to a given prediction under progressively refined constraints [8], [12], [45], [46]; similarly, in robustness radius computation, verification queries are invoked iteratively to narrow down the maximum safe perturbation radius around a given input [19], [27], [36]. Such analyses naturally give rise to sequences of closely related verification queries that differ in limited aspects of their specifications, such as refined input domains or strengthened output constraints. Despite this, current verification tools do not explicitly exploit this structural similarity: each query is restarted from scratch, and information derived during previous runs is discarded.

In this work, we seek to mitigate these inefficiencies by reusing lemmas derived from earlier verification runs to accelerate verification of subsequent queries. Similar incremental solving techniques have proven successful in SAT and SMT solving [6], [14], [17]. Our approach is inspired by CDCL and incremental SAT, where learned conflicts are reused, and we adapt and extend this idea to neural network verification. In the case of neural network verification, previous work has considered proof transfer for abstract-interpretation-based methods [40] and warm-starting branch-and-bound by heuristically resuming the search from leaf nodes of search trees generated from prior solver runs [37], [39]. However, none of the previous work has explored reusing lemmas across multiple invocations of branch-and-bound-based complete verifiers, which is the focus of this work.

Our incremental verification approach is designed for sequences of closely related properties on the same neural network. Our approach records conflicts that arise during branch-and-bound verification, where each conflict captures an infeasible combination of branching decisions. These conflicts are preserved beyond individual verification runs and reused in subsequent queries with refined specifications. We formally define a refinement condition for the conflict to remain valid and show that this condition can be established in several important applications of neural network verification. The inherited conflicts allow the verifier to immediately prune previously explored infeasible regions, avoiding redundant

https://doi.org/

analysis and computation.

Following this, we develop a framework for conflict recording and sound reuse that integrates directly with branch-and-bound-based verifiers, and employ a SAT solver to efficiently manage and apply large collections of learned conflicts during solving. To investigate the effectiveness of the proposed approach, we instantiate it in the Marabou verifier [26], [28], [43] and perform a thorough evaluation on three representative verification tasks—robustness radius computation, iterative input splitting, and formal explanation—and demonstrate consistent empirical reductions in verification runtime, with speedups of up to $1.9\times$ compared to the non-incremental baseline.

The rest of the paper is organized as follows. Section II introduces the necessary background on neural network verification, branch-and-bound search, case splitting, and bound propagation. Section III presents our incremental verification framework, including conflict clauses, query refinement, and the sound reuse of learned conflicts, and describes the implementation details of the proposed approach. Section IV evaluates the approach on several verification tasks and discusses the experimental results. Section V reviews related work. Finally, Section VI outlines directions for future work, and Section VII concludes the paper.

II. Background

A. Neural Networks and the Verification Problem

Neural Networks. A deep neural network is a sequence of L layers, where layer i contains $d^{(i)}$ neurons, with $i = 0$ denoting the input layer and $i = L$ the output layer. Such a network defines a function $f : \mathbb{R}^{d^{(0)}} \to \mathbb{R}^{d^{(L)}}$. Each layer applies an affine transformation followed by an activation function [22]. In this work, we focus on networks with ReLU activations, $\mathrm{ReLU}(t) = \max(0, t)$, though the approach extends naturally to other piecewise-linear activations.

For each layer i, the computation is given by $z^{(i)} = \mathbf{W}^{(i)} x^{(i-1)} + \mathbf{b}^{(i)}$ and $x^{(i)} = \mathrm{ReLU}(z^{(i)})$, where $x^{(0)} = x_0$ is the network input. We denote by $z_j^{(i)}$ and $x_j^{(i)}$ the pre- and post-activation values of the j-th neuron in layer i. Each ReLU neuron is associated with a Boolean phase variable $r_j^{(i)}$, where $r_j^{(i)} = \top$ corresponds to the active phase ($z_j^{(i)} \geq 0$) and $r_j^{(i)} = \bot$ to the inactive phase ($z_j^{(i)} < 0$).

Verification Queries. Neural network verification asks whether a network can exhibit undesired behavior. A verification query for network $f : \mathbb{R}^{d^{(0)}} \to \mathbb{R}^{d^{(L)}}$ is defined by a pair $(\mathcal{X}, \mathcal{Y})$, where $\mathcal{X} \subseteq \mathbb{R}^{d^{(0)}}$ and $\mathcal{Y} \subseteq \mathbb{R}^{d^{(L)}}$ specify input and output regions, respectively, and asks whether $\exists x \in \mathcal{X}$ such that $f(x) \in \mathcal{Y}$. Typically, both regions are described by linear constraints, with $\mathcal{Y}$ representing an undesired output set. A query is SAT if such an input exists and UNSAT otherwise.

B. Branch-and-Bound Verification

Most modern neural network verifiers combine case splitting with bound propagation, a paradigm known as branch-and-bound, to reason about the disjunctive behavior induced by ReLU activations [5], [11], [16], [26], [28], [38], [42], [50]. In this work, we focus on the case-splitting component.

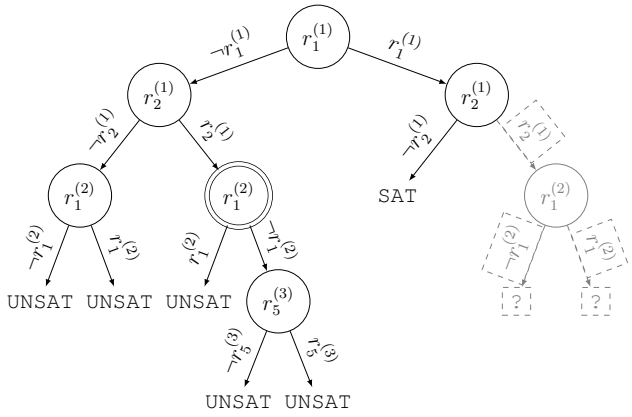

Fig. 1: Branch-and-bound search tree over ReLU phase decisions.

Case Splitting. Verification of ReLU neural networks relies on case splitting over ReLU activation phases. Each split fixes a single phase variable $r_j^{(i)}$ to either the active or inactive case, i.e., $r_j^{(i)} = \top$ or $r_j^{(i)} = \bot$, thereby refining the original verification query into independent subproblems.

Branch-and-Bound Search and Search Tree. Given a neural network f and a verification query $q = (\mathcal{X}, \mathcal{Y})$, branch-and-bound verification explores the space of ReLU activation phases induced by q via repeated case splitting, inducing a search tree (Figure 1). The order in which nodes are split depends on the verifier's heuristic and is not necessarily in numerical order.

The root of the tree corresponds to the original query with no fixed phase decisions. Each node is associated with a *partial assignment*, also called a *decision trail*, $\pi = \{\ell_1, \ldots, \ell_k\}$, where each literal $\ell_i \in \{r_j^{(i)}, \neg r_j^{(i)}\}$ fixes the activation phase of a ReLU neuron. The decision trail π defines a refined subproblem obtained by conjoining the corresponding phase constraints with the original query, and edges correspond to individual case splits extending π by one additional literal. Figure 1 uses $r_1^{(1)}, r_2^{(1)}, r_1^{(2)}, r_5^{(3)}$ to denote phase variables. For example, the double-circled node in Figure 1 corresponds to the decision trail $\pi = \{\neg r_1^{(1)}, r_2^{(1)}\}$. At each node, the verifier applies bound propagation to compute over-approximations of neuron pre- and post-activation values. If these bounds imply that no input consistent with π can satisfy the query, then π is infeasible and the node is declared UNSAT and its subtree is pruned. If a concrete input $x \in \mathcal{X}$ consistent with π satisfies $f(x) \in \mathcal{Y}$, the node is declared SAT (e.g., the $\pi = \{r_1^{(1)}, \neg r_2^{(1)}\}$ leaf in Figure 1), after which the search terminates and unexplored branches need not be visited.

The search terminates when either a SAT leaf is found or all explored leaves are UNSAT. In the latter case, no violating input exists and the property is verified. This branch-and-bound paradigm underlies modern neural network verifiers [11], [26], [28], [38], [42], [50].

Bound Propagation. Bound propagation analyzes the feasibility of subproblems arising during branch-and-bound search (i.e., nodes in the search tree) by computing over-approximations of neuron values under a partial assignment π [34], [41], [47]. These bounds detect infeasible assignments and implied activation phases, enabling early pruning of the search tree (e.g., the UNSAT leaves in Figure 1).

III. Incremental Verification via Learned Conflicts

A. Conflicts and Query Refinement

During branch-and-bound verification, infeasible subproblems encountered along the search can be recorded and reused in the form of conflict clauses.

Definition 1 (Conflict Clause)**.** *A conflict clause for a verification query q is a set of literals $c = \{\ell_1, \ldots, \ell_k\}$ such that the corresponding CNF clause $(\neg\ell_1 \vee \neg\ell_2 \vee \cdots \vee \neg\ell_k)$ is logically implied by q.*

When a decision trail $\pi = \{\ell_1, \ldots, \ell_k\}$ is found to be UNSAT, the infeasible combination can be summarized by the conflict clause $(\neg\ell_1 \vee \cdots \vee \neg\ell_k)$.

To enable sound reuse of conflict clauses across verification queries, we formalize when one query is a *refinement* of another.

Definition 2 (Query Refinement)**.** *A verification query q_2 is a* refinement *of another query q_1, denoted $q_2 \preceq q_1$, if both queries are defined over the same network f and*

$$\mathcal{X}(q_2) \subseteq \mathcal{X}(q_1) \quad \textit{and} \quad \mathcal{Y}(q_2) \subseteq \mathcal{Y}(q_1).$$

We say that a query q_2 *inherits* from q_1 if $q_2 \preceq q_1$, in which case conflicts learned for q_1 may be reused when solving q_2. Intuitively, q_2 imposes stricter constraints than q_1, and therefore admits a smaller feasible region. In particular, any conjunction of constraints that is infeasible under q_1 remains infeasible under q_2. This monotonicity is formalized in Lemma 1.

In practice, many verification workflows progressively strengthen the input domain while keeping output constraints unchanged, as in the cases below. Our framework also supports refinements of both domains. More generally, the soundness argument only requires the feasible set of the later query to be contained in that of the earlier query, and therefore also applies when refinement is introduced through additional conjunctive constraints, such as fixed activation phases.

B. Soundness of Conflict Reuse

Lemma 1 (Monotonicity of Infeasibility Under Refinement)**.** *Let q_1 and q_2 be verification queries such that $q_2 \preceq q_1$, and let $\{\ell_1, \ldots, \ell_k\}$ be a set of literals. If the subproblem $q_1 \wedge \ell_1 \wedge \cdots \wedge \ell_k$ is infeasible, then the subproblem $q_2 \wedge \ell_1 \wedge \cdots \wedge \ell_k$ is also infeasible.*

Lemma 1 formalizes the monotonicity of infeasibility under query refinement. The proof is provided in Appendix A.

Theorem 1 (Sound Conflict Reuse under Refinement)**.** *Let q_1 and q_2 be verification queries such that $q_2 \preceq q_1$. Let $c = \{\ell_1, \ldots, \ell_k\}$ be a conflict clause for q_1. Then c is also a conflict clause for q_2.*

Theorem 1 follows directly from Lemma 1 and establishes the soundness of conflict inheritance across refined queries. A detailed proof is given in Appendix A. As a result, conflicts learned for one query can be reused to prune infeasible regions in refined queries without re-exploration.

C. Using Conflicts during Verification

Having established the soundness of conflict reuse under query refinement, we now describe how conflict clauses are used during verification to prune the search space. Specifically, we employ a SAT solver to reason about inherited conflict clauses during branch-and-bound search as an additional pruning and propagation step.

Checking Consistency via SAT. With each verification query, the verifier inherits a set of conflict clauses $\mathcal{C}$ learned from previous related queries. These clauses are encoded once as CNF clauses in a SAT solver.

At each node of the branch-and-bound search tree, after standard bound propagation is applied, the verifier invokes the SAT solver to check the consistency of the current partial assignment with the inherited conflict clauses $\mathcal{C}$. Let α denote the current partial assignment to ReLU phase variables. Each literal $\ell \in \alpha$ is asserted as a unit assumption in the SAT solver.

SAT-based reasoning under assumptions can yield two outcomes: either the current trail is UNSAT, certifying the subproblem as infeasible, or unit propagation derives implied assignments that further restrict the search space. An UNSAT result precludes any extension of the current partial assignment, while implied assignments enforce necessary literals without excluding feasible solutions.

Lemma 2 (Soundness of SAT-Based Pruning)**.** *Let α be a partial assignment and $\mathcal{C}$ a set of conflict clauses implied by a verification query q. If the propositional CNF formula constructed from α and $\mathcal{C}$ is UNSAT, then no extension of α can correspond to a feasible solution for q.*

Lemma 3 (Soundness of SAT-Based Implied Assignments)**.** *Let α be a partial assignment and $\mathcal{C}$ a set of conflict clauses implied by a verification query q. Let α' be the partial assignment obtained by SAT-based reasoning over α and $\mathcal{C}$, and let $\mathcal{L} = \alpha' \setminus \alpha$ denote the implied assignments. Then any feasible solution for q extending α must satisfy all literals in $\mathcal{L}$.*

Together, Lemmas 2 and 3 establish that SAT-based reasoning over inherited conflict clauses enables both sound pruning and sound propagation, reducing the search space while preserving correctness. Full proofs are provided in Appendix B.

D. Incremental Verification Workflow and Integration

This subsection describes the end-to-end workflow by which incremental conflict reuse is integrated into branch-and-bound verification. At a high level, verification proceeds over a sequence of related queries, where conflicts learned during earlier runs are recorded and selectively inherited by subsequent

Algorithm 1: ICA: Incremental Conflict Analyser

Fields:

POOL: map $id \mapsto$ set of conflicts $\mathcal{C}_{id}$
SAT: SAT solver instance

Interface: The component exposes three procedures: BEGINQUERY($\mathcal{I}$), PROPAGATE($Bounds$), and RECORDCONFLICT(id, c).

Function BEGINQUERY($\mathcal{I}$)
// $\mathcal{I}$: set of inherited identifiers

1 SAT.RESET() // fresh SAT instance for this query
2 **foreach** $id' \in \mathcal{I}$ **do**
3 **foreach** $c \in$ POOL$[id']$ **do**
4 SAT.ADDCLAUSEASCNF(c)

Function PROPAGATE($Bounds$)
// $Bounds$: current variable bounds

5 $\alpha \leftarrow$ EXTRACTPARTIALASSIGNMENT($Bounds$)
6 $res \leftarrow$ SAT.SOLVEUNDERASSUMPTIONS(α)
7 **if** $res =$ *UNSAT* **then**
8 **return** (*UNSAT*, $\emptyset$)
9 $\Delta_{\text{sat}} \leftarrow$ SAT.GETUNITIMPLIEDLITERALS()
10 $Bounds$.APPLYIMPLIEDLITERALS(Δ_{sat})
11 **return** (*SAT*, $Bounds$)

Function RECORDCONFLICT(id, c)
// id: query identifier
// c: conflict clause

12 **if** $\exists c' \in$ POOL$[id]$ *such that* $c' \subseteq c$ **then**
13 **return**
14 POOL$[id] \leftarrow$ POOL$[id] \cup \{c\}$
15 SAT.ADDCLAUSEASCNF(c)

queries. Algorithm 1 presents the Incremental Conflict Analyser component that manages conflict storage and SAT-based reasoning, while Algorithm 2 shows how the component is invoked within the branch-and-bound search loop.

Incremental Conflict Analyser. The core component enabling incremental conflict reuse is the *Incremental Conflict Analyser* (ICA), shown in Algorithm 1. The ICA is responsible for storing, retrieving, and applying learned conflict clauses across related verification queries.

Each verification query q is issued together with an *inheritance set* $\mathcal{I}$ of query identifiers whose learned conflicts may be reused. The ICA maintains a global pool of conflict clauses indexed by query identifier, along with a SAT solver instance populated with the conflicts active for the current query.

At the start of each verification, ICA.BEGINQUERY is invoked (Lines 1–4) to reset the SAT solver and load all conflict clauses associated with the identifiers in $\mathcal{I}$. During branch-and-bound search, ICA.PROPAGATE performs SAT-based reasoning over the current subproblem by passing the current partial assignment over ReLU phase variables as assumptions to the SAT solver (Lines 5–11). If the SAT instance is UNSAT, this result is reported back to the verifier for pruning; otherwise, implied ReLU phase assignments are translated into the corresponding ReLU bound constraints and passed to the verifier's standard propagation. Finally, ICA.RECORDCONFLICT records newly discovered conflicts (Lines 12–15), making them available for reuse in subsequent refined queries. Before storing a conflict c, it checks whether an existing conflict $c' \subseteq c$ already subsumes it. This case arises when conflict propagation detects that the current trail violates an inherited conflict and reports it as UNSAT. Although this result reaches the conflict-recording path, the trail does not represent a new conflict: its infeasibility was established by a conflict already stored in the incremental component. Recording the larger trail is therefore unnecessary. This redundancy check prevents unnecessary growth of the conflict pool and SAT clause database and is not required for soundness.

Branch-and-Bound with Incremental Reuse. Algorithm 2 presents a standard branch-and-bound verification procedure augmented with incremental conflict reuse; incremental extensions are highlighted in blue. At the start of verification, the Incremental Conflict Analyser is initialized for the current query by invoking ICA.BEGINQUERY($\mathcal{I}$) (Line 1), which activates the conflicts inherited from prior queries.

The main search loop follows the standard branch-and-bound structure. For each decision trail π, numeric bound propagation is applied first (Line 6). If a counterexample is found, the procedure terminates with SAT. If the subproblem is infeasible, EXTRACTCONFLICT records the complete activation-phase decision trail π as a conflict. The conflict is then stored via ICA.RECORDCONFLICT, after which the branch is pruned.

If numeric propagation is inconclusive, the verifier invokes incremental conflict reasoning by calling ICA.PROPAGATE (Line 14). This step checks the current partial assignment against inherited conflict clauses. If the SAT solver reports UNSAT, the node is immediately pruned; otherwise, any implied ReLU phase assignments are applied as additional propagation step. The solver then selects an undecided ReLU phase and branches using the existing branching heuristic.

The proposed incremental conflict reuse mechanism is a lightweight, sound extension to branch-and-bound verification that preserves the solver's core reasoning while reducing redundant exploration. It leaves the verifier's numeric reasoning and branching machinery unchanged, adding only conflict-based pruning and phase propagation. Thus, even as the underlying branch-and-bound backend becomes faster, conflict reuse can provide additional benefits by avoiding repeated exploration of infeasible activation-phase combinations. Because its conflict reasoning operates independently, it integrates naturally with any branch-and-bound verifier, including those using GPU-accelerated propagation or parallel search.

Algorithm 2: Branch-and-Bound Verification with Incremental Conflict Reuse

Input: Verification query q on network f;
query identifier id; inherited identifier set $\mathcal{I}$;
Incremental Conflict Analyser object ICA.
Output: SAT, UNSAT, or TIMEOUT.

```
ICA.BeginQuery(I)
π ← ∅                          // Current decision trail
Q ← stack containing π
while Q ≠ ∅ do
    π ← Q.Pop()
    (status, Bounds) ← Propagate(q, π)
     // Standard bound propagation
    if status = SAT then
        return SAT  // Counterexample found
    if status = UNSAT then
        // Record conflict
        c ← ExtractConflict(π)
        ICA.RecordConflict(id, c)
        continue
    else                                   // UNKNOWN
        // Incremental conflict
           reasoning
         (icaStatus, Bounds) ←
                              ICA.Propagate(Bounds)
        if icaStatus = UNSAT then   // Violates
         inherited conflict
            continue
    r ← ChooseSplit(Bounds)      // Split on
     an undecided ReLU phase
    Q.Push(π ∪ {r})
    Q.Push(π ∪ {¬r})
return UNSAT
```

IV. Incremental Verification Use Cases

In this section, we first summarize our implementation and evaluation metrics. We then consider three use cases of incremental verification to evaluate the proposed conflict-reuse mechanism. For each, we explain how its structure induces related queries, show that refinement relations follow from this structure (Propositions 1–3), and compare incremental performance against a non-incremental baseline.

Implementation Details

Our implementation integrates the Incremental Conflict Analyser into the Marabou verifier [28], [43], using the CaDiCaL SAT solver [9] for conflict reasoning. All verification queries induced by the same task share a single ICA instance.

Propagation from Inherited Conflicts. Inherited conflict clauses influence verification in two ways. SAT-based reasoning may detect that the current partial assignment is infeasible, allowing the verifier to prune the subproblem immediately, or it may derive additional ReLU phase assignments via unit propagation. These implied assignments introduce additional linear constraints that are integrated into the verifier's existing numeric reasoning, further restricting the feasible region. To quantify the impact of conflict inheritance, we measure the total number of such effects—both pruned subproblems and implied assignments—over the course of a verification task.

We now turn to the tasks considered in our evaluation.

A. Use Case 1: Determining the Local Robustness Radius

In this common use case, the goal is to identify the largest neighborhood around a reference input within which a neural network's output remains consistent. Given a reference input x_0 and a task-specific notion of output consistency (e.g., preservation of the predicted class or bounded deviation of the output), the goal is to determine tight lower and upper bounds on the maximal robustness radius, up to a specified precision.

We formalize this task via *local robustness verification queries* and the associated *local robustness radius*.

Local robustness verification queries given ε. In the classification setting considered here, a robustness query asks whether the predicted class changes within a bounded perturbation region around a reference input x_0. Given a norm $\|\cdot\|_p$ and a radius $\varepsilon > 0$, the network is *not locally robust* at x_0 with respect to ε if there exists an input x within the ε-ball around x_0 that violates the desired output property. Formally, let $\mathcal{P}(x_0, x)$ denote a task-specific predicate capturing misclassification, the network is not locally robust if

$$\exists x \in \mathbb{R}^n \text{ such that } \|x - x_0\|_p \leq \varepsilon \ \wedge \ \mathcal{P}(x_0, x).$$

Definition 3 (Local Robustness Radius)**.** *Let $f : \mathbb{R}^n \to \mathbb{R}^m$ be a neural network and $x_0 \in \mathbb{R}^n$ a reference input. Let $\mathcal{P}(x_0, x)$ denote a task-specific predicate expressing output inconsistency. The* local robustness radius *at x_0 is defined as*

$$\varepsilon^\star \ = \ \inf \{\varepsilon \geq 0 \mid \exists x \in \mathbb{R}^n, \ \|x - x_0\| \leq \varepsilon \ \wedge \ \mathcal{P}(x_0, x)\}.$$

The task is to compute certified bounds $\underline{\varepsilon} \leq \varepsilon^\star \leq \overline{\varepsilon}$ such that $\overline{\varepsilon} - \underline{\varepsilon} \leq \delta$ for a given precision $\delta > 0$. The lower bound corresponds to a formally verified robustness guarantee (UNSAT), while the upper bound corresponds to a radius at which a violating input is found (SAT).

Local robustness radius computation is a standard benchmark in neural network verification and provides a quantitative measure of model stability around a given input.

A standard approach to this task repeatedly invokes a neural network verifier over a sequence of verification queries, typically using a binary-search style procedure over candidate perturbation radii. Each query examines a radius ε_i within a prescribed interval: an UNSAT result certifies robustness at ε_i and yields a new lower bound, while a SAT result produces a violating input and yields a new upper bound. The next candidate radius is selected accordingly, and the process continues until the remaining interval is within the desired precision or the computational budget is exhausted. A precise procedural description is given in the extended version [20].

Method	Time (s)	Solved	Propagations	Conflicts
Non-incremental	315.6	160	–	–
Incremental	**233.5**	**185**	8.2	107.4
Speedup	**1.35**×	–	–	–

TABLE I: Robustness radius evaluation on MNIST.

This procedure induces a sequence of verification queries $q_1, q_2, \ldots, q_k$, all defined over the same network f and reference input x_0. Each query q_i corresponds to a perturbation radius $\varepsilon_i \geq 0$ and checks the output predicate $\mathcal{P}(x_0, x)$ for all inputs satisfying $\|x - x_0\| \leq \varepsilon_i$. Across the sequence, the perturbation radius is the only varying component and determines the input constraint of the i-th query.

Proposition 1 (Robustness Query Refinement)**.** *Let q_i and q_j be two robustness verification queries with perturbation radii $\varepsilon_i > \varepsilon_j$. Then q_j is a refinement of q_i, denoted $q_j \preceq q_i$.*

Proposition 1 shows that robustness radius determination induces a refinement-ordered family of verification queries. The proof is provided in Appendix C.

In the incremental robustness radius determination procedure, each verification query q_i inherits conflicts learned from all previously issued queries q_j with larger perturbation radii $\varepsilon_j > \varepsilon_i$. This allows later queries, which impose stricter input constraints, to reuse conflicts learned under looser constraints and prune infeasible regions early. Since the binary search explores radii in a non-monotonic order, conflicts are selectively inherited only from queries with $\varepsilon_j > \varepsilon_i$, which are guaranteed to be valid refinements by Proposition 1.

Evaluation. To evaluate the effectiveness of incremental verification for determining the local robustness radius, we compared our incremental approach against a non-incremental baseline on 1000 points from the MNIST dataset, using an MNIST fully connected neural network from the VNN-COMP benchmark [4], [13].[1] For each point, we ran the procedure to compute the local robustness radius using precision parameter $\delta = 0.001$.

Table I summarizes the results. The *Time* column reports the average total solving time per task (in seconds), *Solved* denotes the number of inputs for which the robustness radius was determined within the timeout, *Propagations* reports the average number of propagations induced by inherited conflicts per task, and *Conflicts* reports the average number of conflicts recorded per task. Overall, incremental verification with conflict reuse significantly outperforms the baseline. On average, the incremental method achieves a $1.3\times$ speedup in per-task solving time, substantially reducing the total time required to determine robustness radii across test inputs.

Figure 2 presents a visual comparison of runtimes for incremental and non-incremental robustness radius determination. Each point corresponds to a single input instance. While both methods perform similarly on relatively easy queries (e.g., those solved within 600 seconds), a clear gap appears as the queries become harder: the incremental approach solves instances faster and, in some cases, succeeds where the non-incremental baseline times out.

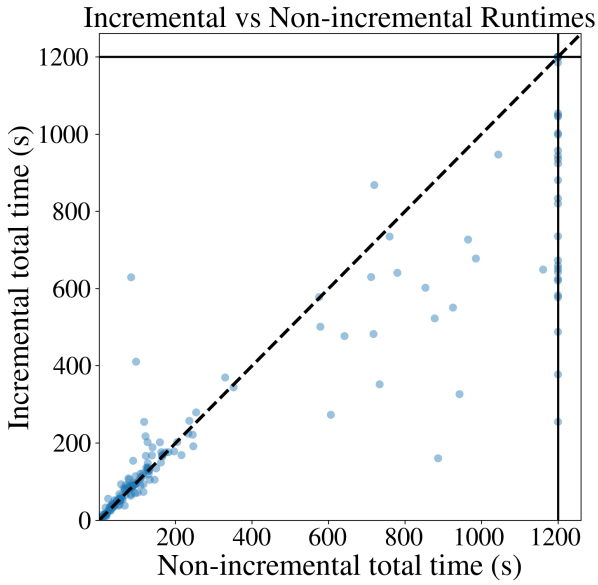

Fig. 2: Incremental vs. non-incremental for robustness radius determination.

To quantify the overhead of SAT-based conflict reasoning, we measured the fraction of the incremental runtime spent in the conflict-reuse component over the full robustness-radius evaluation. For queries completed within 120 seconds, this component accounted for 26.98% of the runtime on average, whereas for harder queries taking more than 120 seconds, it accounted for only 4.90%. Thus, the relative overhead of conflict reasoning is more noticeable on easy queries and becomes small compared with the underlying BaB reasoning on harder queries.

B. Use Case 2: Verification with Input Splitting

Input splitting is a common approach for verifying challenging properties, often referred to as split-and-conquer or divide-and-conquer verification. Under input splitting, the input region is partitioned into multiple subregions, and each subregion is verified independently. As shown by Wu et al. [44], if the verifier is sound and complete and the subregions collectively cover the original input region, the overall verification result can be determined compositionally. That is, the property is SAT if at least one subregion is proven SAT, and UNSAT if all subregions are proven UNSAT.

Input splitting is particularly important for scaling verification to difficult properties where the full input region is too large to verify at once. By recursively partitioning challenging regions into smaller subregions, verifiers can verify properties that would otherwise time out.

With input splitting, each child query produced by splitting is a refinement of the parent query that was split. This follows

[1] https://github.com/VNN-COMP/vnncomp2021/blob/main/benchmarks/mnistfc/mnist-net_256x2.onnx

because the input region of each child query is strictly contained within the input region of the parent query. The following proposition formalizes this observation.

Proposition 2 (Input Splits are Refinements)**.** *Let q_i be a verification query, and suppose $q_{i+1}, \ldots, q_{i+n}$ are generated by input splitting of q_i. Then each $q_{i+1}, \ldots, q_{i+n}$ is a refinement of q_i.*

Proposition 2 shows that input splitting induces refinement chains, enabling incremental verification across recursive partitioning. The proof is provided in Appendix C.

The input-splitting incremental verification procedure leverages Proposition 2 to enable incremental verification over a recursive partitioning of the input space. Given a verification query q with input region $\mathcal{X}$, the verifier first attempts to verify q directly. If verification yields a conclusive result (SAT or UNSAT), it is returned. If verification times out, $\mathcal{X}$ is partitioned by splitting the largest interval dimension at its midpoint, producing two child queries q_{left} and q_{right}.

By Proposition 2, both child queries are refinements of the parent query q. Therefore, all conflict clauses learned during the attempted verification of q are retained and reused when verifying q_{left} and q_{right}. Each child query is then verified recursively following the same procedure: if a child query times out, it is further split and inherits all conflict clauses from its ancestors. As input splitting progresses, the set of learned conflict clauses therefore monotonically increases.

The procedure continues until all leaf subregions are resolved or a time limit is reached. If all leaf subregions are determined UNSAT, the original property is UNSAT. Conversely, if any leaf subregion is determined SAT, the original property is SAT. Pseudocode for the input-splitting workflow appears in the extended version [20].

Evaluation. To evaluate the effectiveness of incremental conflict reuse for input splitting, we applied our approach to a counterexample-guided inductive synthesis (CEGIS) loop for training Lyapunov neural certificates for deep reinforcement learning-controlled spacecraft, as introduced by Mandal et al. [30]. This framework iteratively learns a Lyapunov function that certifies reach-while-avoid properties for a 4D spacecraft docking system. The CEGIS loop alternates between training the certificate to satisfy Lyapunov conditions and formally verifying these conditions using neural network verification. When verification identifies counterexamples that violate the Lyapunov conditions, the counterexamples are added to the training data and the certificate is retrained.

For each input region, we first attempt to verify the region as a whole. If the verifier returns SAT or UNSAT, no further splitting is performed. If the verification attempt times out, the region is split and the same procedure is applied recursively to the resulting child regions.

We executed the complete CEGIS training procedure and extracted the 680 verification queries generated during the process. Of these, 189 queries were solved without requiring input splitting and were excluded from the evaluation because they do not generate the recursive parent-child refinement structure needed for conflict inheritance. The remaining 491 queries required input splitting and form the basis of our evaluation. For these queries, we used the same progressive timeout strategy for both the incremental and non-incremental configurations, following [30]: the initial verification attempt was run with a 5-second timeout, and each subsequent input split increased the timeout by a factor of 1.5 (i.e., 7.5 seconds after the first split, 11.25 seconds after the second, and so on). A global timeout of 1200 seconds was imposed to prevent indefinite splitting.

Table II summarizes the performance of incremental and non-incremental methods. The *Time* column reports average verification time per verification task. *Solved* denotes the number of verification tasks completed within the global timeout. *Propagations* reports the average number of propagations induced per verification task, and *Conflicts* reports the average number of conflict clauses recorded during verification.

Method	Time (s)	Solved	Propagations	Conflicts
Non-incremental	84.1	489	–	–
Incremental	**43.9**	**491**	1.7	7.9
Speedup	**1.92**×	–	–	–

TABLE II: Evaluation of iterative input splitting for Lyapunov certificate verification.

Overall, the incremental method significantly outperforms the non-incremental baseline. It successfully solves all 491 verification tasks, whereas the baseline times out on two tasks, and achieves an average speedup of $1.92\times$. On average, each verification attempt involved 7.9 recorded conflict clauses, which induced 1.7 propagations per attempt. Figure 3 presents a visual comparison of runtimes for both methods on a logarithmic scale. Each point corresponds to a single verification query. Most points lie below the equal-performance line, indicating that incremental verification outperforms the non-incremental baseline on the majority of queries.

C. Use Case 3: Minimal Sufficient Feature Set Extraction

Minimal sufficient feature set extraction addresses the problem of explaining a neural network's prediction by identifying a smallest subset of input features whose values alone suffice to determine the output [8], [12], [31], [45], [46].

Given a reference input and its predicted class, the goal is to identify a subset of input indices such that fixing these features to their reference values guarantees preservation of the predicted class, regardless of how the remaining features vary. While this notion naturally extends to regression settings via a tolerance on the output, we focus here on classification for clarity.

Formally, we seek a feature subset that is sufficient for preserving the prediction and minimal with respect to set inclusion, subject to a given time budget. Unfixed features are allowed to vary freely over their entire domain.

Definition 4 (Minimal Sufficient Feature Set)**.** *Let $f : \mathbb{R}^n \to \mathbb{R}^m$ be a neural network, $x_0 \in \mathbb{R}^n$ a reference input, and*

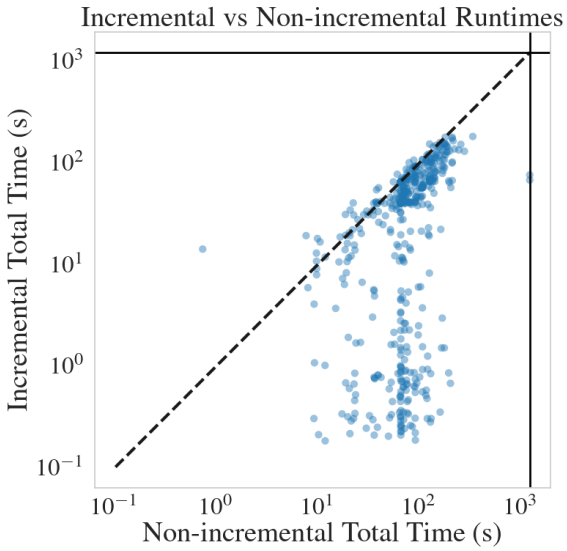

Fig. 3: Incremental vs. non-incremental for input splitting.

$c = \arg\max_j f_j(x_0)$ *the predicted class at* x_0*. For a subset of feature indices* $S \subseteq \{1, \ldots, n\}$*, define*

$$\mathcal{X}_S \;=\; \{x \in \mathbb{R}^n \mid x_i = (x_0)_i \text{ for all } i \in S\}\,.$$

The set S *is a* sufficient feature set *if*

$$\forall x \in \mathcal{X}_S, \quad \arg\max_j f_j(x) = c.$$

It is minimal *if no strict subset* $S' \subset S$ *is sufficient.*

Minimal sufficient feature sets reveal which input features are essential for a given prediction and form a central primitive in formal explainability.

Fig. 4: Minimal sufficient feature set visualizations for GTSRB inputs. Pixels not included in the explanation are shown in gray.

A common approach to solving this task is to search over subsets of input features while invoking a neural network verifier as a subroutine. The procedure begins with all features fixed to their values in the reference input and progressively frees features to test whether the predicted class is preserved. At a high level, the procedure alternates between proposing candidate feature sets to free and using verification outcomes to determine whether they can be freed or must remain fixed. This process follows an *anytime* paradigm: if interrupted, it returns the smallest sufficient feature set found so far.

To explore the space of feature subsets efficiently, an importance ordering over features is typically computed in a preprocessing step and used to guide a binary-search-style exploration. Rather than freeing features individually, the procedure considers sets of features—often starting from the least important ones—and recursively refines them. Candidate sets are tentatively freed and checked for sufficiency; depending on the outcome, features are either permanently freed or selectively reinstated. A precise procedural description of this workflow, including the technical modifications required to support incremental verification and the binary-search strategies employed, is provided in the extended version [20].

Under this formulation, the task gives rise to a collection of verification queries, all defined over the same neural network f and reference input x_0. Each query q is parameterized by a candidate set of freed features $\overline{S}$, inducing the fixed feature set $S = \{1, \ldots, n\} \setminus \overline{S}$ and the corresponding constrained input set $\mathcal{X}_S$. The query asks whether fixing all features in S to their values in x_0 is sufficient to preserve the predicted class c. A query returning UNSAT certifies that S is sufficient and all features in $\overline{S}$ can be freed. In contrast, a SAT result indicates that S is insufficient and that additional features must be fixed. For soundness, TIMEOUT outcomes are treated as SAT.

All queries share the same network f, reference input x_0, and target class c; the only varying component is the set of fixed features. As a result, minimal sufficient feature set extraction induces a family of closely related verification queries whose input constraints differ only in which features are fixed.

Due to the binary-search-style exploration, the verification queries generated during minimal sufficient feature set extraction are naturally organized as a search tree rather than a linear sequence. Queries generated after a SAT or TIMEOUT outcome impose strictly stronger input constraints by fixing additional features. In contrast, queries generated after an UNSAT outcome test disjoint sets of features and are therefore incomparable under refinement. Consequently, the overall query family is partially ordered by refinement.

We formalize the refinement relationship that arises along SAT and TIMEOUT branches of the search tree below.

Proposition 3 (Feature-Set Query Refinement)**.** *Let* q *and* q' *be two verification queries generated during the feature set extraction procedure, corresponding to fixed feature sets* S *and* S'*, respectively. If* q' *is generated from* q *following a SAT or TIMEOUT outcome, then* $S \subset S'$ *and* q' *is a refinement of* q*, denoted* $q' \preceq q$*.*

Proposition 3 shows that, although the full set of queries is not totally ordered, all queries along SAT and TIMEOUT branches form refinement chains. The proof is provided in Appendix C.

Accordingly, in the incremental minimal sufficient feature set extraction procedure, each query q' generated after a SAT or TIMEOUT outcome inherits all conflicts learned from its ancestor queries along the same search-tree branch.

Evaluation. We evaluated the effectiveness of incremental conflict reuse for accelerating minimal sufficient feature set extraction on the German Traffic Sign Recognition Benchmark (GTSRB) dataset [35], using a convolutional neural network model from Wu et al. [45]. Out of the 1000 test inputs

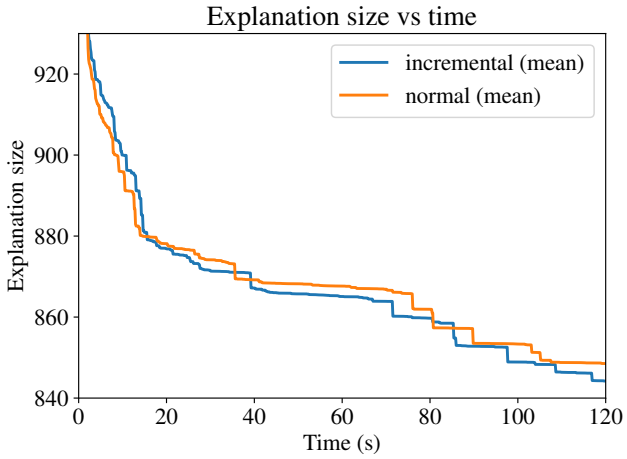

Fig. 5: Incremental vs. non-incremental explanation size over time.

considered, 70 produced SAT or TIMEOUT execution paths in which conflicts were learned from infeasible BaB nodes and could be reused in subsequent refined queries. For the remaining inputs, the procedure did not generate reusable conflicts and therefore could not benefit from conflict-based pruning or propagation. We restrict our conflict-reuse evaluation to these 70 cases.

For each input, we ran an *anytime* minimal sufficient feature set extraction procedure and compared our incremental approach against a non-incremental baseline. Table III summarizes the results. The *Explanation Size* column reports the average size of the minimal sufficient feature set returned within the time budget. Smaller explanations are due to timeouts occurring after more progress. *Propagations* reports the average number of propagations induced by inherited conflicts per task, and *Conflicts* reports the average number of conflict clauses recorded per task. Both approaches achieve comparable explanation sizes, with the incremental method yielding slightly smaller explanations on average. On average, the incremental approach records 92 conflicts per task, which induce approximately 2 effective propagations per task.

Method	Explanation Size	Propagations	Conflicts
Non-incremental	848.52	–	–
Incremental	**844.21**	2.30	92.14

TABLE III: Minimal sufficient feature set extraction on GTSRB.

The primary benefit of incremental verification in this setting lies in its *anytime* behavior. As shown in Figure 5, the two methods perform similarly during the initial phase, up to approximately 20 seconds, with the non-incremental approach sometimes slightly ahead. This reflects the overhead of recording conflicts before enough reusable information has accumulated.

Beyond this point, incremental verification tends to achieve smaller explanations more quickly by reusing previously learned conflicts accumulated earlier in the search. This reuse supports earlier identification of critical features and more efficient pruning of infeasible feature combinations, leading to improved anytime behavior. Overall, these observations suggest that conflict reuse can be beneficial for explainability tasks under an anytime verification regime.

D. Discussion

We evaluated incremental conflict reuse on three verification tasks: robustness radius determination, input splitting, and minimal sufficient feature set extraction.

In robustness radius determination, conflicts learned at larger radii can be reused when verifying smaller radii. However, because binary search does not visit radii in monotonic order, reuse is selective: a query can inherit conflicts only from previously considered queries with larger radii. This yields a 26% reduction in average verification time compared to the non-incremental baseline (from 315.6 to 233.5 seconds), corresponding to a $1.35\times$ speedup.

Input splitting provides the strongest refinement structure among the three tasks. Every child region is a direct refinement of its parent, and conflicts learned in parent and ancestor subproblems can therefore be reused throughout the corresponding recursive split branch. This yields a 47% reduction in average verification time compared to the non-incremental baseline (from 84.1 to 43.9 seconds), corresponding to a $1.92\times$ speedup.

For minimal sufficient feature set extraction, refinement occurs only along branches induced by SAT and TIMEOUT results. An UNSAT result does not induce a subsequent refinement, which limits the number of conflicts that can be inherited. Consequently, this task has the weakest reuse structure of the three. While the final improvement in explanation size is small, incremental reuse improves anytime behavior by producing smaller explanations later in the computation.

The observed results reflect these differences in refinement structure. Input splitting, where every generated child refines its parent, achieves the largest speedup. Robustness radius determination provides fewer opportunities for reuse and achieves a more moderate speedup, while feature set extraction permits reuse only along selected branches and primarily benefits in its anytime behavior. Thus, query families with more frequent refinement relationships enable greater conflict reuse and tend to yield larger performance gains.

V. Related Work

Incremental SAT and SMT solving address scalability by reusing learned information, such as conflict clauses and theory lemmas, across sequences of related problem instances [6], [14], [17]. When successive instances are structurally similar, this can avoid redundant reasoning and improve performance. But, the effectiveness of incremental techniques is strongly problem-dependent, and worst-case results in areas such as dynamic graph algorithms show that substantial recomputation may be unavoidable under general updates [23]. Despite the maturity

of incremental SAT/SMT solving, its systematic application to neural network verification remains limited.

Prior work on incremental neural network verification has largely focused on settings in which the network itself changes across queries. Residual reasoning [18] reuses information learned for abstract networks to accelerate verification after refinement, while IVAN and I-IVAN [37], [39] reuse successful case splits heuristically across related network architectures. More recently, Zhang et al. [48] study incremental verification guided by counterexample potentiality, again in scenarios where the network is modified. De Palma et al. [15] use an IVAN-inspired strategy that reuses branch-and-bound leaf constraints across related queries. In contrast, we reuse activation-phase conflicts whose validity is preserved under query refinement.

Related ideas have been explored in abstract interpretation. FANC [40] employs heuristic transfer of abstract bounds to certify multiple approximate neural networks. In contrast, we consider iterative verification over a fixed neural network, where properties or input constraints vary across queries. Our approach reuses learned conflict clauses and provides conditions under which they can be transferred soundly between related verification queries, aligning with conflict-based incremental SAT/SMT solving applied to a single network.

In our evaluation, we instantiated our approach with the CaDiCaL SAT solver [9] and the Marabou verifier [28], [43] as backends. CaDiCaL is a modern SAT solver with a clean design that facilitates its integration with other tools. Marabou is a proof-producing DNN analysis framework [24], which has been used for a myriad of applications, including network pruning [29], formal explainability [7], verifying network generalization [1], and network ensembles [2]. While our implementation relies on Marabou and CaDiCaL, the proposed technique is solver-agnostic and can, in principle, be integrated with other SAT solvers and branch-and-bound verifiers.

VI. Limitations and Future Work

The current implementation records the complete activation-phase decision trail as a conflict, without attempting to minimize it. As a result, a conflict may include ReLU phase decisions that are not strictly necessary to establish infeasibility. Smaller conflicts could improve reuse effectiveness and reduce the overhead of SAT-based reasoning. We leave the exploration of minimization of such conflicts for future work [25], [49], [51].

Marabou's push/pop interface is used to add and remove query-specific constraints without rebuilding the shared network encoding. However, conflict reuse in our framework remains restricted to monotone refinements. Supporting arbitrary push/pop or assumption-based reuse would require tracking the constraints on which each learned conflict depends, since removing a constraint may invalidate conflicts learned while it was active. We leave such provenance-aware conflict reuse as a promising direction for future work.

More generally, the approach focuses on reusing conflicts derived from infeasible subproblems. Other reusable information, such as richer theory-specific lemmas or abstractions, may further improve performance in some settings. Exploring such mechanisms would require careful consideration of both soundness and solver integration.

Our approach reuses conflicts from infeasible (UNSAT) leaves. Queries known to be UNSAT and their refinements are not issued, so gains arise from inheriting conflicts from SAT or TIMEOUT queries that encounter infeasible leaves. Dually, since satisfying assignments extend to relaxed queries, such queries are not issued. In the refinement direction, however, satisfying assignments may be reused heuristically, but this does not provide sound pruning guarantees. Developing principled ways to leverage SAT outcomes while maintaining such guarantees is an interesting direction for future work.

Finally, learned conflicts are used solely for pruning and propagation. An additional direction is to exploit conflicts to guide branching decisions, for example by prioritizing frequently occurring case splits.

VII. Conclusion

We introduced an incremental verification approach for neural networks that reuses learned conflict clauses across related verification queries over a fixed network. By formalizing a refinement relation between queries, we showed when such conflicts can be reused soundly and integrated this mechanism into the Marabou verifier as a lightweight extension to branch-and-bound search. Our evaluation shows that conflict reuse reduces redundant exploration and yields speedups of up to $1.9\times$ on representative iterative verification tasks, while preserving soundness.

Acknowledgments

Raya Elsaleh is supported by the Ariane de Rothschild Women Doctoral Program. The work of Elsaleh and Katz was partially funded by a grant from the Israeli Science Foundation (grant number 558/24), and 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. Wu's work is supported by a gift from the VMware University Research Fund.

Appendix A
Soundness of Conflict Reuse

This appendix contains the proofs of Lemma 1 and Theorem 1 from Section III.

A.1 Proof of Lemma 1

Proof. Assume for contradiction that $q_2 \wedge \ell_1 \wedge \cdots \wedge \ell_k$ is feasible, and let x be a satisfying input. Since $q_2 \preceq q_1$, both queries are defined over the same network f, $\mathcal{X}(q_2) \subseteq \mathcal{X}(q_1)$, and the ReLU phase literals $\ell_1, \ldots, \ell_k$ refer to the same ReLU constraints in both queries. Therefore, the same input x satisfies $q_1 \wedge \ell_1 \wedge \cdots \wedge \ell_k$, contradicting infeasibility.

□

A.2 Proof of Theorem 1

Proof. Let $c = \{\ell_1, \ldots, \ell_k\}$ be a conflict clause for q_1. By Definition 1, the subproblem $q_1 \wedge \ell_1 \wedge \cdots \wedge \ell_k$ is infeasible. Since $q_2 \preceq q_1$, infeasibility is monotone under refinement by Lemma 1. Therefore, the subproblem $q_2 \wedge \ell_1 \wedge \cdots \wedge \ell_k$ is also infeasible, and hence c is a conflict clause for q_2.

□

Appendix B
Soundness of SAT-Based Conflict Reasoning

This appendix provides proofs for Lemmas 2 and 3, which establish the soundness of SAT-based pruning and propagation using learned conflict clauses.

B.1 Proof of Lemma 2

Proof. Suppose, for contradiction, that there exists a feasible solution for query q that extends the partial assignment α. Then there exists a complete assignment that satisfies all literals in α and corresponds to a feasible input for q.

Since the propositional CNF formula constructed from α and the clauses in $\mathcal{C}$ is UNSAT, every complete assignment extending α must violate at least one conflict clause $c \in \mathcal{C}$. By Definition 1, each conflict clause forbids the simultaneous satisfaction of all its literals, and any assignment violating c cannot correspond to a feasible solution for q. This contradicts the assumption of feasibility. Therefore, no extension of α can correspond to a feasible solution for q.

□

B.2 Proof of Lemma 3

Proof. Let $\mathcal{L}$ be the set of literals implied by unit propagation when solving the SAT instance under assumptions α. By definition of unit propagation, for each $\ell \in \mathcal{L}$, assigning $\neg\ell$ under α would falsify at least one conflict clause $c \in \mathcal{C}$.

Suppose, for contradiction, that there exists a feasible solution for q that extends α but violates some implied literal $\ell \in \mathcal{L}$. Then the corresponding complete assignment satisfies $\alpha \cup \{\neg\ell\}$ and therefore violates the conflict clause c. By Definition 1, this assignment cannot correspond to a feasible solution for q, contradicting feasibility.

Hence, every feasible extension of α must satisfy all literals in $\mathcal{L}$.

□

Appendix C
Proofs of Refinement Properties of Verification Use Cases

This appendix provides proofs of the refinement properties for the verification use cases discussed in Section IV.

C.1 Proof of Proposition 1

Proof. The two queries are defined over the same network f and reference input x_0. The input constraints of the two queries are given by

$$\mathcal{X}(q_i) \;=\; \{x \in \mathbb{R}^n \mid \|x - x_0\| \le \varepsilon_i\}\,,$$
$$\mathcal{X}(q_j) \;=\; \{x \in \mathbb{R}^n \mid \|x - x_0\| \le \varepsilon_j\}\,.$$

Since $\varepsilon_j < \varepsilon_i$, it follows immediately that $\mathcal{X}(q_j) \subseteq \mathcal{X}(q_i)$. The output constraints of both queries are identical, as they are defined by the same predicate $\mathcal{P}(x_0, x)$. Therefore, by Definition 2, q_j is a refinement of q_i.

□

C.2 Proof of Proposition 2

Proof. Let q_0 be a verification query, and let q_1 be a verification query generated by input splitting of q_0. By construction, input splitting partitions the input domain of q_0 into subregions, and thus the input domain of q_1 is a subset of that of q_0, i.e., $\mathcal{X}(q_1) \subseteq \mathcal{X}(q_0)$. Moreover, input splitting does not modify the network or the output constraints. Hence, $\mathcal{Y}(q_1) = \mathcal{Y}(q_0)$, and in particular $\mathcal{Y}(q_1) \subseteq \mathcal{Y}(q_0)$. Therefore, by Definition 2, q_1 is a refinement of q_0.

□

C.3 Proof of Proposition 3

Proof. By construction, a successor query q' generated after a SAT or TIMEOUT outcome is obtained by reinstating a subset of previously freed features. Thus, the corresponding fixed feature sets satisfy $S \subset S'$.

The input constraints of the two queries are given by

$$\mathcal{X}_S = \{x \in \mathbb{R}^n \mid x_i = (x_0)_i \text{ for all } i \in S\}\,,$$
$$\mathcal{X}_{S'} = \{x \in \mathbb{R}^n \mid x_i = (x_0)_i \text{ for all } i \in S'\}\,.$$

Since $S \subset S'$, it follows that $\mathcal{X}_{S'} \subseteq \mathcal{X}_S$.

Both queries enforce the same output constraint, namely preservation of the predicted class c. Therefore, by Definition 2, q' is a refinement of q.

□

References

[1] G. Amir, O. Maayan, T. Zelazny, G. Katz, and M. Schapira. Verifying Generalization in Deep Learning. In *Proc. 35th Int. Conf. on Computer Aided Verification (CAV)*, pages 438–455, 2023.

[2] G. Amir, T. Zelazny, G. Katz, and M. Schapira. Verification-Aided Deep Ensemble Selection. In *Proc. 22nd Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 27–37, 2022.

[3] D. Amodei, C. Olah, J. Steinhardt, P. F. Christiano, J. Schulman, and D. Mané. Concrete Problems in AI Safety, 2016. Technical Report. http://arxiv.org/abs/1606.06565.

[4] 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.

[5] S. Bak, H.-D. Tran, K. Hobbs, and T. T. Johnson. Improved Geometric Path Enumeration for Verifying ReLU Neural Networks. In *Proc. 32nd Int. Conf. on Computer Aided Verification (CAV)*, pages 66–96, 2020.

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

[7] S. Bassan, G. Amir, D. Corsi, I. Refaeli, and G. Katz. Formally Explaining Neural Networks within Reactive Systems. In *Proc. 23rd Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 10–22, 2023.

[8] S. Bassan and G. Katz. Towards Formal XAI: Formally Approximate Minimal Explanations of Neural Networks. In *Proc. 29th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 187–207, 2023.

[9] 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.

[10] M. Bojarski, D. W. del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba. End to End Learning for Self-Driving Cars, 2016. Technical Report. http://arxiv.org/abs/1604.07316.

[11] E. Botoeva, P. Kouvaros, J. Kronqvist, A. Lomuscio, and R. Misener. Efficient Verification of ReLU-Based Neural Networks via Dependency Analysis. In *Proc. 34th AAAI Conf. on Artificial Intelligence (AAAI)*, pages 3291–3299, 2020.

[12] R. Boumazouza, R. Elsaleh, M. Ducoffe, S. Bassan, and G. Katz. FAME: Formal Abstract Minimal Explanation for neural networks. In *Proc. 14th Int. Conf. on Learning Representations (ICLR)*, 2026.

[13] C. Brix, M. N. Müller, S. Bak, T. T. Johnson, and C. Liu. First Three Years of the International Verification of Neural Networks Competition (VNN-COMP). *International Journal on Software Tools for Technology Transfer (STTT)*, 25(3):329–339, 2023.

[14] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proc. 14th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.

[15] A. De Palma, G. Dolcetti, and C. Urban. Faster Verified Explanations for Neural Networks. In *Proc. 40th European Conf. on Object-Oriented Programming (ECOOP)*, 2026.

[16] H. Duong, T. Nguyen, and M. B. Dwyer. NeuralSAT: A High-Performance Verification Tool for Deep Neural Networks. In *Proc. 37th Int. Conf. on Computer Aided Verification (CAV)*, pages 409–423, 2025.

[17] N. Eén and N. Sörensson. An Extensible SAT-Solver. In *Proc. 7th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT)*, pages 502–518, 2004.

[18] Y. Y. Elboher, E. Cohen, and G. Katz. Neural Network Verification Using Residual Reasoning. In *Proc. 20th Int. Conf. on Software Engineering and Formal Methods (SEFM)*, pages 173–189, 2022.

[19] Y. 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 IEEE/ACM Digital Avionics Systems Conf. (DASC)*, pages 1–6, 2024.

[20] R. Elsaleh, L. Davis, H. Wu, and G. Katz. Incremental Neural Network Verification via Learned Conflicts, 2026. Technical Report. http://arxiv.org/abs/2603.12232.

[21] A. Esteva, B. Kuprel, R. A. Novoa, J. Ko, S. M. Swetter, H. M. Blau, and S. Thrun. Dermatologist-Level Classification of Skin Cancer with Deep Neural Networks. *Nature*, 542(7639):115–118, 2017.

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

[23] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-Logarithmic Deterministic Fully-Dynamic Algorithms for Connectivity, Minimum Spanning Tree, 2-Edge, and Biconnectivity. *Journal of the ACM (JACM)*, 48(4):723–760, 2001.

[24] 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.

[25] 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.

[26] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*, pages 97–117, 2017.

[27] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Towards Proving the Adversarial Robustness of Deep Neural Networks. In *Proc. 1st Workshop on Formal Verification of Autonomous Vehicles (FVAV)*, pages 19–26, 2017.

[28] G. Katz, D. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljić, D. Dill, M. Kochenderfer, and C. Barrett. The Marabou Framework for Verification and Analysis of Deep Neural Networks. In *Proc. 31st Int. Conf. on Computer Aided Verification (CAV)*, pages 443–452, 2019.

[29] O. Lahav and G. Katz. Pruning and Slicing Neural Networks using Formal Verification. In *Proc. 21st Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 183–192, 2021.

[30] U. Mandal, G. Amir, H. Wu, I. Daukantas, F. L. Newell, U. Ravaioli, B. Meng, M. Durling, M. Ganai, T. Shim, G. Katz, and C. Barrett. Formally Verifying Deep Reinforcement Learning Controllers with Lyapunov Barrier Certificates. In *Proc. 24th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 95–106, 2024.

[31] J. Marques-Silva and A. Ignatiev. Delivering Trustworthy AI through Formal XAI. In *Proc. 36th AAAI Conf. on Artificial Intelligence (AAAI)*, pages 12342–12350, 2022.

[32] C. Păsăreanu, D. Gopinath, and H. Yu. Compositional Verification for Autonomous Systems with Deep Learning Components. In H. Yu, X. Li, R. M. Murray, S. Ramesh, and C. J. Tomlin, editors, *Safe, Autonomous and Intelligent Vehicles*, pages 187–197. Springer, 2019.

[33] M. Sälzer and M. Lange. Reachability is NP-Complete Even for the Simplest Neural Networks. In *Proc. 15th Int. Conf. on Reachability Problems (RP)*, pages 149–164, 2021.

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

[35] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel. The German Traffic Sign Recognition Benchmark: A Multi-Class Classification Competition. In *Proc. IEEE Int. Joint Conf. on Neural Networks (IJCNN)*, pages 1453–1460, 2011.

[36] C. Strong, H. Wu, A. Zeljić, K. Julian, G. Katz, C. Barrett, and M. Kochenderfer. Global Optimization of Objective Functions Represented by ReLU Networks. *Machine Learning*, 112(10):3685–3712, 2023.

[37] X. Tang. Improved Incremental Verification for Neural Networks. In *Proc. 18th Int. Conf. on Theoretical Aspects of Software Engineering (TASE)*, pages 392–409, 2024.

[38] H. Tran, X. Yang, D. Manzanas Lopez, P. Musau, L. Nguyen, W. Xiang, S. Bak, and T. Johnson. NNV: The Neural Network Verification Tool for Deep Neural Networks and Learning-Enabled Cyber-Physical Systems. In *Proc. 32nd Int. Conf. on Computer Aided Verification (CAV)*, pages 3–17, 2020.

[39] S. Ugare, D. Banerjee, S. Misailovic, and G. Singh. Incremental Verification of Neural Networks. In *Proc. 44th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2023.

[40] S. Ugare, G. Singh, and S. Misailovic. Proof Transfer for Fast Certification of Multiple Approximate Neural Networks. In *Proc. 37th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2022.

[41] 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 (USENIX Security)*, pages 1599–1614, 2018.

[42] S. Wang, H. Zhang, K. Xu, X. Lin, S. Jana, C.-J. Hsieh, and J. Kolter. Beta-CROWN: Efficient Bound Propagation with Per-Neuron Split Constraints for Complete and Incomplete Neural Network Verification. In *Proc. 35th Conf. on Neural Information Processing Systems (NeurIPS)*, 2021.

[43] 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)*, pages 249–264, 2024.

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

[45] M. Wu, X. Li, H. Wu, and C. Barrett. Efficiently Computing Compact Formal Explanations, 2025. Technical Report. http://arxiv.org/abs/2409.03060.

[46] M. Wu, H. Wu, and C. Barrett. VeriX: Towards Verified Explainability of Deep Neural Networks. In *Proc. 37th Conf. on Neural Information Processing Systems (NeurIPS)*, pages 22247–22268, 2023.

[47] K. Xu, Z. Shi, H. Zhang, Y. Wang, K.-W. Chang, M. Huang, B. Kailkhura, X. Lin, and C.-J. Hsieh. Automatic Perturbation Analysis for Scalable Certified Robustness and Beyond. In *Proc. 34th Conf. on Neural Information Processing Systems (NeurIPS)*, 2020.

[48] G. Zhang, Z. Zhang, H. Bandara, S. Chen, J. Zhao, and Y. Sui. Efficient Incremental Verification of Neural Networks Guided by Counterexample Potentiality. In *Proc. 40th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2025.

[49] H. Zhang, S. Wang, K. Xu, L. Li, B. Li, S. Jana, C.-J. Hsieh, and J. Kolter. General Cutting Planes for Bound-Propagation-Based Neural Network Verification. In *Proc. 36th Conf. on Neural Information Processing Systems (NeurIPS)*, 2022.

[50] H. Zhang, T.-W. Weng, P.-Y. Chen, C.-J. Hsieh, and L. Daniel. Efficient Neural Network Robustness Certification with General Activation Functions. In *Proc. 32nd Conf. on Neural Information Processing Systems (NeurIPS)*, pages 4939–4948, 2018.

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