NDD: A Decision Diagram for Network Verification

Zechun Li*, Peng Zhang*, Yichi Zhang*, Hongkun Yang†
*Xi'an Jiaotong University, †Google

Abstract

State-of-the-art network verifiers extensively use Binary Decision Diagram (BDD) as the underlying data structure to represent the network state and equivalence classes. Despite its wide usage, we find BDD is not ideal for network verification: verifiers need to handle the low-level computation of equivalence classes, and still face scalability issues when the network state has a lot of bits.

To this end, this paper introduces *Network Decision Diagram (NDD)*, a new decision diagram customized for network verification. In a nutshell, NDD wraps BDD with another layers of decision diagram, such that each NDD node represents a *field* of the network, and each edge is labeled with a BDD encoding the values of that field. We designed and implemented a library for NDD, which features a native support for equivalence classes, and higher efficiency in memory and computation. Using the NDD library, we re-implemented five BDD-based verifiers with minor modifications to their original codes, and observed a 100× reduction in memory cost and 100× speedup. This indicates that NDD provides a drop-in replacement of BDD for network verifiers.

1 Introduction

Network verification is becoming an important toolchain for network operators to ensure the correctness of their networks [9, 10, 13, 14, 16, 19, 24–28, 30, 32–34, 37, 39, 41, 47, 48, 50–58], and some of them have been deployed by large service providers to detect misconfigurations [32, 37, 48, 54].

Many network verifiers use *Binary Decision Diagram (BDD)* [11, 20], to compactly represent network state like packet headers, routes, failures, etc. Despite its extensive usage, we find that BDD is not ideal for network verification tasks, in the following three aspects.

Memory inefficiency. We observe BDD-based network verifiers create a lot number of redundant nodes, which share the same sub-structure on some fields, but cannot be reduced since their differences on other fields. This redundancy is magnified when verifiers [13, 57] compute equivalence classes (i.e., atomic predicates, or simply atoms [1]), which leads to an "explosion of atoms" and overflows the memory.

Computation inefficiency. We observe that BDD-based network verifiers are slowed down by logical operations, e.g., $a \wedge b$. The reason is BDD operations proceed in a recursive way, with each recursion looking at a single bit of a and b. Even worse, if the BDDs encode different variables, more recursions are needed to "align" their variables. Considering networks often match on multiple fields (e.g., 5-tuple, 104 bits), the recursions can be deep and therefore very slow.

Lack of support for network verification. We observed that BDD libraries offer little support for common tasks of network verification, e.g., computation of atoms [51], incremental update of atoms [13, 57], and handling of packet transformers [53]. As a result, network verifiers need to design their own algorithms, which can be quite complex, for these common tasks, and still encounter scalability problems due to the aforementioned explosion of atoms.

We attribute the above problems to the fact that BDD is agnostic of *fields*, an important semantics in networks, where a device processes routes and packets by matching one or multiple fields. Moreover, the matching shows a characteristic of "field locality": *each rule matches few but different fields of the network.* This makes the fields mostly orthogonal, such that the number of nodes can be a cross-product of that for each field, and therefore quite large.

Based on the above observation, our basic idea is: *instead of viewing the entire network state as a whole, we partition it into a set of* ***fields***, so that logical operations are restricted to BDDs of the same field, and redundant nodes of the same fields can be reduced, fully independent of other fields.

After decoupling fields, a question arise: how to represent network state with the per-field BDDs? A naive way of using conjunctions of per-field BDDs is not compact, and can lead to an explosive number of conjunctions. Our approach to the

[1] For brevity, the following of this paper will use the term "atoms" to refer equivalence classes or atomic predicates [51].

representation problem is to wrap the per-field BDDs with another layer of decision diagram, termed *Network Decision Diagram (NDD)*. In an NDD, each NDD node u encodes a field f of multiple bits, and each edge of u is labeled with a BDD of field f. We show such a representation is *compact*, and prove it is actually *canonical*. In addition, since NDDs can look at many bits each time, logical operations of NDDs require much less recursions.

Finally, instead of computing single set of atoms for all fields, we compute a set of atoms for each field f, and transform the labels of each NDD edge from BDDs into a set of atoms. We term the resultant NDD as *atomized NDD*. Then, logical operations (e.g., $\wedge$, $\vee$) on NDDs can be replaced with simpler set operations (e.g., $\cap$, $\cup$), similar to [51].

The atomized NDD offers a lot of benefits. First, since the atoms are over a single field, we avoid the cross-product effect and can dramatically reduce the total number of atoms in the network. Moreover, all atom-related operations can be *fully transparent* to network verifiers, such that verifiers are agnostic of atoms, and do not need to realize their own atom-related algorithms.

Based on the above ideas, we realize an NDD library [2]. It features a native support for the computation and incremental update of atoms, and provides an efficient way for handling packet transformers. We use the NDD library to re-implement 5 BDD-based network verifiers, i.e., AP Verifier [52], APT [53], APKeep [57], SRE [58], and Batfish [19], with minor modifications to their original codes. After replacing the BDD library with the NDD library, these verifiers show a 2 orders-of-magnitude reduction in both memory and time.

Contributions. In sum, this paper makes three contributions:

- We introduce Network Decision Diagram (NDD), a new decision diagram customized for network verification.
- We implement an NDD library, and use it for 5 different network verifiers with small modified LOCs.
- We use real and synthetic datasets to show that by using NDD, the 5 verifiers achieve a 2 orders-of-magnitude reduction in both memory and time.

Limitations. NDD builds on the assumption of field locality, which generally holds on our datasets. However, if rules in a network match most of the fields, NDD may not perform better than BDD (§6.6).

2 Motivation

2.1 Preliminary to BDD

Binary Decision Diagram (BDD) is a rooted, directed acyclic graph (DAG) with two terminal nodes (`true` and `false`), and multiple non-terminal (variable) nodes (refer to Figure 1 as an example). Each non-terminal node u represents a test of a boolean variable $var(u)$, and has two outgoing edges, i.e., a high edge and low edge (shown as solid and dashed line, respectively). These two edges point to two *successor* nodes $high(u)$ and $low(u)$, i.e., a high successor and a low successor, respectively. The semantics is that: if x is tested to be *true* (*false*), then the high (low) edge will be followed and the next test will be $high(u)$ ($low(u)$). Each path from the root node to the terminal node `true` corresponds to one truth assignment of a boolean formula. For the BDD in Figure 1, there are only two paths from the root node to the terminal node `true`, encoding two truth assignments of $x_1 = false, x_2 = true$, and $x_1 = true, x_2 = true, x_3 = true$, respectively. Therefore, this BDD represents a boolean formula $(\neg x_1 \wedge x_2) \vee (x_1 \wedge x_2 \wedge x_3)$.

[2]open sourced at https://github.com/XJTU-NetVerify/NDD

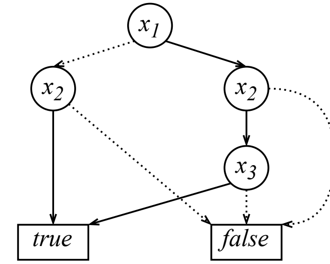


Figure 1: An example of BDD encoding the boolean formula $(\neg x_1 \wedge x_2) \vee (x_1 \wedge x_2 \wedge x_3)$.

Reduced Ordered BDD. A BDD is said to be an *Ordered* (OBDD) if the boolean variables follow a fixed variable order (say $x_1 < x_2 < \ldots, < x_n$ where $x_i < x_j$ means variable x_i appears before variable x_j) on all paths through the graph. An OBDD is said to be a *Reduced* (ROBDD) if it satisfies the following two conditions:

(C1) Uniqueness: no two distinct nodes represent the same variable and have the same successors;

(C2) No redundant node: no non-terminal node has identical high and low successor.

An important property of ROBDD is *canonicity*, i.e., for any boolean formula, if the variable ordering is specified, there is a unique ROBDD representing it [11]. As a result, ROBDD can significantly reduce the node redundancy [20] for a compact representation of boolean formulas, and is very efficient for logical operations. Without specified otherwise, the BDDs in this paper are ROBDDs.

BDD library. To use BDD, applications like network verifiers leverage off-the-shelf BDD libraries [7, 8]. A BDD library needs to realize logical operations, e.g., `and`, `or`, `not`, and other useful functions like `exist`, `restrict`, `oneSat`, `satCount`, etc. Most of them are realized in a recursive way.

To achieve high efficiency of memory and time, modern BDD libraries rely on some efficient algorithms and data structures. The *node table (unique table)* is a hash table holding all BDD nodes, each of which is a distinct tuple $(var, low, high)$. When the node table is full, a BDD library will free unused nodes with *garbage collection (gc)*. A common way to realize

BDD

```
1. foreach 1≤i≤32: vars(i) ← bdd.createVar();
2. foreach d ∈ devices: Preds.add(calPortPred(d, vars));
3. foreach pred ∈ Preds:
4.   foreach a ∈ Atoms:
5.     set.add(bdd.and(a, pred));
6.     set.add(bdd.and(a, bdd.not(pred)));
7.   Atoms ← set;
8. foreach d ∈ devices, p ∈ d.ports:
9.   foreach a ∈ Atoms:
10.    if (bdd.and(p.pred,a) = a):
11.      p.atoms.add(a);
12. traverseRec(Atoms, src);

13. Function traverseRec(pktSet, d)
14.   if (!pktSet.isEmpty()):
15.     foreach p ∈ d.ports:
16.       forwarded ← pktSet.intersect(p.atoms);
17.       traverseRec(forwarded, p.nextHop);
```

NDD

```
1. var ← ndd.createVar(32);
2. foreach d ∈ devices: Preds.add(calPortPred(d, var));
3. ndd.atomize(Preds);
4. traverseRec(ndd.true, src);

5. Function traverseRec(pktSet, d)
6.   if (pktSet != false):
7.     foreach p ∈ d.ports:
8.       forwarded ← ndd.and(pktSet, p.pred);
9.       traverseRec(forwarded, p.nextHop);
```

Figure 2: Pseudo code to verify networks based on BDD and NDD, respectively.

gc is to maintain a reference count for each node, and when the reference count of a node reaches zero, the node is marked as dead, and will be freed when gc is called. Most BDD libraries use *operation cache* to speed up logical operations: before each operation, the library firstly checks whether the result exists in the operation cache, and returns the result if there is a match; otherwise, the operation is performed and the result is put into the cache.

2.2 The use of BDD in network verification

In the recent decade, BDD has been extensively used for network verification. For example, many data plane verifiers [13, 51, 57] use BDDs to efficiently represent packet equivalent classes, and some control plane verifiers like SRE [57] use BDD to represent failure equivalence classes. Interested readers can refer to Appendix §G for an incomplete list of network verifiers based on BDDs.

The top of Figure 2 shows a code snippet to implement data plane verification based on BDD [51]. Note that we over-simplify the codes to reflect the core logic. First, the verifier creates a BDD variable for each bit of the field. Then, it computes port predicates for each device (Lines 1-2), computes atoms of all port predicates(Lines 3-7), and compute the set of atoms for each port (Lines 8-11). Finally, it computes reachability by recursively traversing the network with a packet set initialized to all atoms, and at each node filters some of atoms based on the forwarding rules (Lines 12-17).

Despite its extensive usage in network verification, we find BDD, however, is not ideal for achieving high efficiency for either memory or computation. To show this, we use a toy example with four nodes, as shown in Figure 3. For simplicity of illustration, we assume there are four fields, i.e., `dstIP`, `dstPort`, `srcIP`, and `srcPort`, where `srcIP` has only four bits and other fields each has only two bits.

Memory inefficiency. First, we observe that there are a lot of redundant BDD nodes that cannot be reduced. Figure 4(a) shows three BDDs encoding three sets of packets that are reachable from A to port 1 of D, following three different paths (i.e., $A \rightarrow B \rightarrow D$, $A \rightarrow C \rightarrow D$, and $A \rightarrow D$). These three BDDs have a common sub-structure consisting of 14 nodes which encode variables x_1 to x_6 of `srcIP` and `srcPort` (inside the red dashed box). That is, two copies of the sub-structure ($14 \times 2 = 28$ nodes) are redundant. However, we cannot eliminate these 28 nodes. The reason is that their nodes encoding x_6 point to different successor nodes encoding x_7 (the first bit of `dstIP`), and therefore satisfy the uniqueness condition of ROBDD and cannot be represented with a single node. We term such a redundancy as *partial redundancy* as it only exhibits on a subset of variables. The number of 28 redundant nodes may not seem a big issue in this toy example, while for real networks, this number can be as large as $O(10^8)$, leading to a huge memory cost (§6.5).

In addition, the effect of partial redundancy is magnified by the computation of atoms. When computing atoms, a verifier needs to perform many logical conjunctions on BDDs, thereby creating even more redundant nodes for each field. On our virtualized datacenter network datasets, which have multiple layers of packet headers, the number of redundant nodes can overflow the BDD node table, a phenomenon which we term as *explosion of atoms*. Interested readers can refer to Appendix D for an example.

Computation inefficiency. In addition to the high memory cost due to redundant nodes, we observe BDD-based network verifiers are also unnecessarily slow due to the frequent but inefficient logical operations. Specifically, to compute atoms or check reachability, network verifiers need to perform many logical operations ($\wedge$, $\vee$, $\neg$). Inside a BDD library, these operations are realized recursively: a conjunction $a \wedge b$, where a and b encode the same variable, boils down to two recursive calls of $low(a) \wedge low(b)$ and $high(a) \wedge high(b)$. Since each recursion only proceeds one bit, the recursions can go very deep when there are many bits (e.g., 104 bits for 5-tuples).

Worse still, if a and b encode different variables, with an ordering say $var(a) < var(b)$, many recursions may be needed to "align" variables, i.e., recursively find a descendant node of a, say a', such that $var(a') = var(b)$. For example, in Figure 4(a), to compute `and` of BDDs labeled by $A \rightarrow B \rightarrow D$ and b_3, we first need to recursively find the node of b_1, which encodes the same variable x_7 as b_3, rather than directly jumping to the field `dstIP`. This amounts to 17 recursions in total.

One may wonder whether we can mitigate these limitations with a better variable ordering. Putting aside finding an optimal variable ordering is a co-NP-complete problem [20], we observe that the redundancy is there no matter what the ordering of BDD variable is: when we re-order the variables, the redundancy is just shifting from one place to another (see Appendix A.9 for some numerical results.)

Lack of support for network verification. As a general-purpose data structure, BDD offers little support for some important tasks of network verification, including: (1) the computation of atoms, (2) the incremental update of atoms, and (3) the handling of packet transfomers.

First, many verifiers compute atoms over the network state for a faster verification. For example, BDD-based data plane verifiers like AP Verifier [52] partition the header space into a small set of atoms, such that packets in the same atomic predicate have the same forwarding behavior. When computing reachable packet sets from source to destination, the verifiers can use set operations ($\cup$, $\cap$) on sets of atoms, instead of the more expensive logical operations ($\vee$, $\wedge$) on BDDs. However, using BDD, these verifiers need to implement their own algorithms to compute atoms, which is nontrival and leads to explosion of atoms.

Secondly, to avoid from-scratch re-computation of atoms after each network update, network verifiers [13, 30, 57] need to design efficient algorithms to incrementally update the atoms. For example, APKeep [57], a state-of-the-art data plane verifier, designs efficient algorithms to achieve a realtime update of atoms; Katra [13] extends APKeep to multi-layer networks by computing *partial equivalence classes*, which can significantly improve the scalability of data plane verification for multi-layer networks. However, both APKeep and Katra need to implement their own algorithms for updating the set of atoms, which tend to be more complicated than the from-scratch computation of atoms, as in AP Verifier [51].

Finally, packet transformers such as NAT, IP-in-IP [40], VXLAN [38], etc., are common in networks. The existence of packet transformers makes the computation of atoms more complicated. Verifiers like APT [53] and APKeep [57] use customized algorithms to compute the atoms, such that the input and output of any transformer can be represented by a set of atoms. Such algorithms are also complex and may end up with many atoms.

3 Introducing NDD

3.1 Observations

We observe the inefficiency of BDDs in network verification is mainly due to the fact that BDD, as a general-purpose data structure, lacks the notion of "field", an important semantics in networks. For example, network devices forward or filter packets by matching one or multiple packet fields. Without such a notion, network verifiers based on BDDs model the

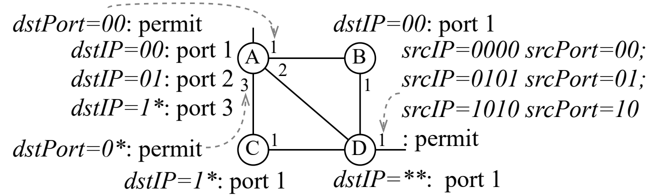


Figure 3: A toy example network used throughout this paper.

network state as a whole, and the number of BDD nodes can grow exponentially with the number of fields, due to the following "field locality" phenomenon.

Field locality refers to the fact that even though there can be many fields in the network, each rule only matches on one or few of these fields, and different rules may match different fields. For example, forwarding rules in the underlay of a network virtualized with VXLAN only match on the `dstIP` of the outer header, while forwarding rules in the overlay only match the `dstIP` of the inner header. As another example, most ACL rules match a single or rather few fields, e.g., only 25 of 392 ACLs in the Purdue campus network match more than one field [46]. Finally, each route policy filters routes only based on one or two of all attributes, e.g., prefix, communities, AS path.

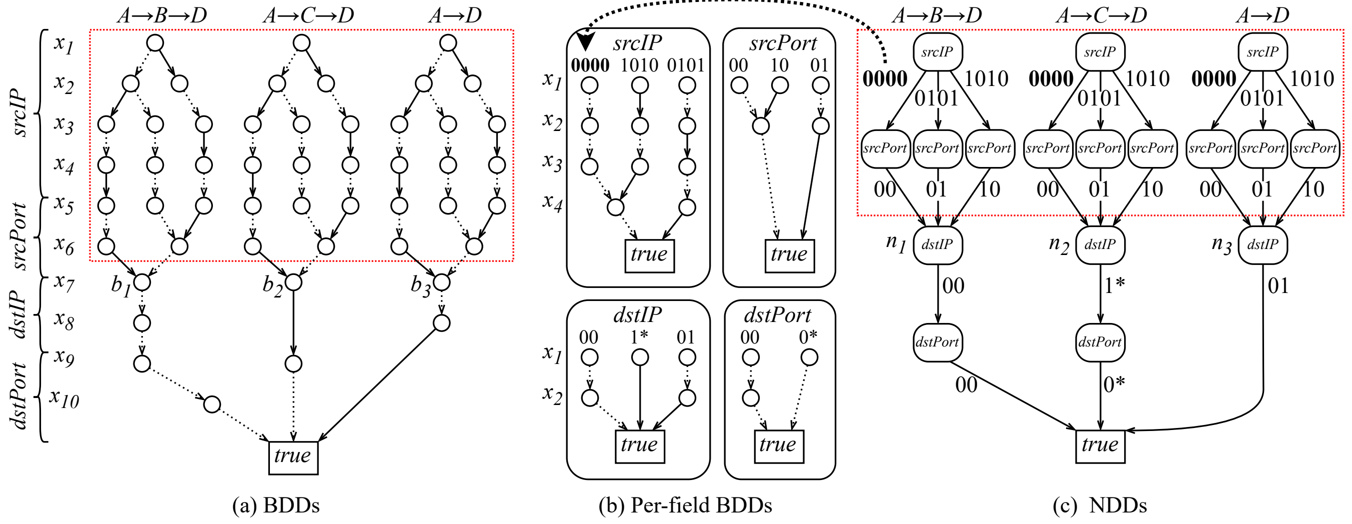
Due to field locality, the different fields in network state are *orthogonal*, meaning that the values of each field tend to be independent of those of other fields. As shown in Figure 4(a), the nodes for `src IP/Port` fields are orthogonal to those of the `dst IP/Port` fields. As a result, multiple copies of the same sub-structure for `src IP/Port` need to be created, due to the different sub-structures for `dst IP/Port`.

3.2 Idea

Decoupling the fields. Our basic idea is partitioning the whole network state (a bit vector) into a set of *fields*, which can be packet headers (e.g., 5-tuple), route attributes (e.g., community, AS Path), or up/down state for different sets of links, etc. Then, for each field, we create per-field BDD nodes and place them into their own *namespace*, such that logical operations and reduction of redundant nodes are restricted to BDDs in the same namespace. As shown in Figure 4(b), after reducing the redundant nodes for each of the four fields, there are only 11, 5, 5 and 3 BDD nodes, respectively. This amounts to a total number of 24 BDD nodes, instead of 50 ones as previously shown in Figure 4(a).

After the decoupling of fields, we can effectively reduce the number of redundant nodes. The question is how can we represent the network state with these BDDs, each of which only encodes a field? A straightforward way is to use a set of conjunctions or arrays of BDDs, where each BDD in an array encodes a specific field (see Appendix E for details). Such an approach is simple but can lead to an explosion of arrays after logical operations (see Appendix A.10 for details).

Assembling fields. We approach this representation problem



(a) BDDs (b) Per-field BDDs (c) NDDs

Figure 4: Illustrations of BDDs and NDDs. The dashed/solid lines represent `false`/`true`, and all edges to the BDD/NDD terminal node `false` are omitted for simplicity.

by layering another layer of decision diagram on top of the per-field BDDs. We term this layer of decision diagram as *Network Decision Diagram (NDD)*. In an NDD, each NDD node represents a field of multiple bits and has multiple edges; each edge of it is labeled with a BDD encoding those bits.

Such a representation is more compact than using BDD arrays or conjunctions. Returning to the example, Figure 4(c) shows the NDDs representing three sets of packets reachable from A to D. Compared to (a) BDDs, NDD only creates one copy of per-field BDD nodes in (b), as shown inside the red box. This leads to 24 BDD nodes in total, instead of 50 ones as in (a). Note that NDD also needs three copies for the same sub-structure; otherwise, it cannot distinguish the three different packet sets. However, since each NDD variable represents a field, each copy of the sub-structure has only 4 NDD nodes, which is much less than 14 BDD nodes as in (a) (in red rectangles).

Moreover, logical operations on NDDs is much faster than those on BDDs, since it can quickly skip a field in one recursion. For example, in Figure 4(c), to compute `and` of the NDD nodes labeled by $A \to B \to D$ and n_3, from the root node of $A \to B \to D$, NDD only needs 6 recursion for all paths in total to reach the node n_1, which has the same variable with n_3. This contrasts with BDDs, which need 17 recursions.

Finally, recall that network verifiers compute atoms over the whole network state, leading to an explosion of atoms. In addition, network verifiers need to design and implement their own algorithm, in order to efficiently compute and update the atoms, which are non-trival tasks.

Atomizing fields. Towards this problem, we design efficient algorithm for the computation and update of atoms as an internal process of NDD. Specifically, we compute a separate set of atoms for each field, and then transform the label of each NDD edge from a BDD to a set of atoms for that field. After that, the logical operations (e.g., conjunctions) on BDDs become set operations (e.g., intersections), which tend to be faster. By computing atoms over individual fields, NDDs avoid the problem of explosion of atoms. We term the NDDs where labels of edges are represented with atoms as *atomized NDDs*. Since NDD natively supports atomization, network verifiers become fully agnostic of atoms, and can still use the same logical operators after the NDDs are atomized.

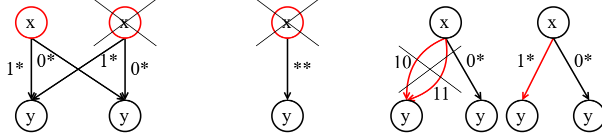
3.3 Network Decision Diagram

Similar to BDD, an Network Decision Diagram (NDD) is also a rooted, directed acyclic graph with two terminal NDD nodes `true` and `false`. The key difference is that there are multiple non-terminal NDD nodes, each of which represents a *field*, rather than a *bit*. Formally, we have the following definition, which is adapted from the definition of BDD in [11].

Definition 1. *A **Network Decision Diagram (NDD)** is a rooted, directed acyclic graph with*

- *two terminal NDD nodes `true` and `false`, with an out-degree of zero.*
- *a set of non-terminal NDD nodes. Each node u is associated with a variable $var(u)$ representing a **field** of one or multiple bits, and has a set of outgoing edges, denoted as $edges(u)$. Each $e \in edges(u)$ points to a successor of u, denoted as $next(e)$, and has a predicate over the variable $var(u)$, denoted as $label(e)$.*
- *$\forall u$, $\forall x, y \in edges(u)$ with $x \neq y$: $label(x) \wedge label(y) = false$, and $\bigvee_{e \in edges(u)} label(e) = true$.*

Note above the third condition generalizes that of BDD, saying that the labels of edges from the same NDD node are exclusive, and the union of these labels covers all values of



(a) Non-unique nodes (b) Redundant nodes (c) Redundant edges

Figure 5: Three types of redundancy in NDD.

the field. For BDD, a label is either 0 or 1, clearly satisfying this condition.

Reduced Ordered NDD. The definition of Reduced Ordered NDD (RONDD) is similar to that of ROBDD (§2.1). The difference is that besides the two conditions *uniqueness* and *no redundant node*, RONDD has a third condition:

(C3) No redundant edges: no two edges from the same NDD node point to the same successor, i.e.,

$$\forall u, \forall x, y \in edge(u) : next(x) = next(y) \Rightarrow x = y \quad (1)$$

Figure 5 shows examples of the three conditions for RONDD. Note that the number of edges for a non-terminal NDD node u can be up to 2^n in the worst case, where n is the number of bits for $var(u)$. But due to the third condition (C3), the actual number is much smaller than that. For example, 99% of NDD nodes in SRE have no more than 10 edges (see Appendix A.6 for details). Similar to ROBDD, RONDD also has the canonicity property, such that RONDD can compactly represent equivalent network state.

Lemma 1. ***Canonicity of NDD.*** *For any boolean function $F(x_1, x_2, \cdots, x_N)$ over N boolean variables with ordering $x_1 < x_2 < \cdots < x_N$, there is exactly only one RONDD u with k NDD variables and ordering $f_1 < f_2 < \cdots < f_k$, where f_i has n_i bits $x_i^1 < x_i^2 < \cdots < x_i^{n_i}$, and $N = \sum_{i=1}^{k} n_i$, such that u can represent F.*

Proof. The proof can be found in Appendix C. □

3.4 Atomized Network Decision Diagram

Atomized Network Decision Diagram is an NDD where the label of each edge is a set of atoms, instead of a BDD.

Definition 2. *Given a set of NDDs $\mathcal{N}$ defined over a set of variables F, we say $\mathcal{A}(f) = \{a_1^f, \dots, a_k^f\}$ is the* ***set of atoms*** *for variable $f \in F$, with respect to $\mathcal{N}$, if it satisfies the following conditions:*

(1) $a_i^f \neq false, \forall i \in \{1, \dots, k\}$;

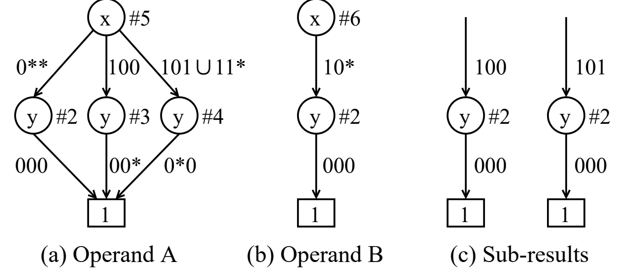
(2) $\vee_{i=1}^{k} a_i^f = true$;

(3) $a_i^f \wedge a_j^f = false$, if $i \neq j$;

(4) $\forall e \in edges(u)$, u is a node of $\mathcal{N}$ with $var(u) = f$: there exists a set $atoms(e) \subset \mathcal{A}(f)$, s.t., $label(e) = \bigvee_{a \in atoms(e)} a$;

(5) k is the minimum number satisfy the above properties.

Definition 3. *Given a set of NDDs $\mathcal{N}$, we say $\mathcal{N}^a$ is the* ***atomized NDDs*** *of $\mathcal{N}$, if $\mathcal{N}^a = \mathcal{N}$ except that for each edge e of $\mathcal{N}$, $label(e) \leftarrow atoms(e)$.*



(a) Operand A (b) Operand B (c) Sub-results

Figure 6: Examples for `and` operation of NDD.

4 Design of the NDD Library

Realizing an efficient NDD library to replace the current BDD libraries for network verifiers is a non-trivial task, given that modern BDD libraries apply various optimizations to reduce memory cost and speed up computation. In this paper, we design an NDD library based on existing BDD libraries like JDD, by (1) extending the standard APIs with new functions, e.g., `atomize`, and `update`; (2) adapting the design of logical operations, garbage collection, and node table of BDDs to work for NDDs, (3) re-using some optimizations of BDD, e.g., operation cache, and (4) applying some new NDD-specific optimizations like removing the `false` NDD node.

4.1 The APIs

Table 1 shows the APIs of our NDD library, which is quite similar to those of BDD libraries: The NDD library provides operations similar to BDD, including `createVar`, `apply`, `not` and `exist`, etc., such that existing verifiers can use the same operators when switching from BDD to NDD. One exception is `createVar`, which takes a parameter of *len* to represent the number of bits for the NDD variable. To enable a native support for the computation and update of atoms, the NDD library also provides two additional functions, `atomize` and `update`, using which a verifier can instruct the NDD library to compute and update the atoms, respectively.

`createVar`. This API lets verifiers to declare a variable in NDD, to represent a field in network. When a verifier calls `createVar` with parameter *len*, the NDD library firstly calls the `createVar` function of BDD libraries to create *len* BDD variables, and then creates an NDD variable based on these *len* BDD variables. Then, the NDD library maintains the mapping between the NDD variable and the set of BDD variables, which helps find the related BDD namespace of an NDD node during NDD operations.

`apply`. This API provides a unified logical operation $op \in \{and, or, diff\}$ on two NDDs. In BDD libraries, `apply` is realized in a recursive way. Taking $c = \texttt{apply}(\texttt{op}, a, b)$ for example, where a and b encode the same BDD variable *var*, the function will recursively compute $l = \texttt{apply}(\texttt{op}, low(a), low(b))$, and $h = \texttt{apply}(\texttt{and}, high(a), high(b))$, and returns a BDD node c with $var(c) = var$, $low(c) = l$ and $high(c) = h$.

Table 1: The APIs of NDD library. The first four are similar to BDD libraries, and the last two are introduced by NDD.

Function	Description
`createVar` (Integer *len*)	create an NDD variable with *len* bits
`apply` (*op*, NDD *a*, NDD *b*)	apply a logical operation *op* on two NDDs *a* and *b*, where $op \in \{and,\ or,\ diff\}$
`not` (NDD *a*)	apply a logical negation on an NDD *a*
`exist` (NDD *a*, Field *field*)	apply existential quantification for field *field* on NDD *a*
`atomize` (Set<NDD> *N*)	compute per-field atoms for a set of NDDs *N*, replace BDDs of edges in *N* with a set of atoms
`update` (NDD δ, NDD *a*)	update the set of atoms when a new NDD δ is added; *a* is an atomized NDD satisfying that $\delta \Rightarrow a$

For NDD, things become more complicated since each NDD node may have multiple successors, each with a predicate. We use the example in Figure 6 to illustrate this. Suppose there are 2 fields x and y each with 3 bits, and we want to compute $\texttt{apply}(\texttt{op}, \#5, \#6)$. Without loss of generality, we assume `op=and`. Unlike in BDD, where both BDD nodes have the same number of 2 successors, the first NDD #5 has 3 successors while the second NDD #6 has a singleton successor. Therefore, we need to enumerate all pairs of successors (one from #5 and the other from #6). If the two successors have overlapping test conditions, we can recursively call `apply` on these two successors; otherwise, we continue to another pair. For this example, the test condition of the left successor of #5 does not overlap with that of #6, since $\texttt{bdd_and}(0{*}{*}, 10{*}) = false$. Therefore, we continue to the middle successor of #5, whose test condition overlaps with that of #6. Then, we compute $\texttt{apply}(\texttt{and}, \#3, \#2)$, and the result is the NDD node #2, and the test condition for this node is 100, as shown on the left of Figure 6(c). Similarly for the right successor of #5, we have the result shown on the right of Figure 6(c). As we can see, there are two edges both pointing to the NDD node #2, with different conditions 100 and 101. Then, NDD can merge these two edges into a single one, and compute a disjunction of the two conditions as $10*$, resulting in the NDD node #6. Thus, we have $\texttt{apply}(\texttt{and}, \#5, \#6) = \#6$.

Theoretically, if two NDD nodes have m and n successors, respectively, we need to enumerate $m \times n$ pairs of successors, and for each pair, we need to compute a conjunction of two BDDs. However, in practice, this number is small. For example, 99% of NDD nodes have less than 10 successors, when we use NDD-based SRE [58] on an 80-nodes fattree networks (Appendix A.6).

`atomize`. This API takes a set of NDDs as input, and computes atoms for each variable, and transforms each of the NDDs into atomized NDDs, where each edge has a set of atoms $atoms(e)$, satisfying $\vee_{a \in atoms(e)} a = label(e)$. We realize this by firstly dividing all the NDD nodes and their successors into groups, one for each NDD variable. After that, for each group, we compute the atoms over all NDD nodes in the group, in a similar way as [51]. Finally, for each edge $e \in edges(u)$ of some NDD node u, we compute $atoms(e)$ as follows: for each atom a of $var(u)$, if $a \Rightarrow label(e)$, we insert a into the atom set $atoms(e)$. For logical operations on two atomized NDDs, we can use set operations over $atoms(e)$ instead of logical operations over $label(e)$.

`update`. This API takes an NDD δ and an atomized NDD a satisfying $\delta \Rightarrow a$ as input, updates per-field atoms and computes the atom set $atoms(e)$ for each edge e of δ and all (direct or indirect) successors of δ. For the NDD node δ, we can compute the conjunctions of BDDs of edges of δ with each atom of field $var(\delta)$. But this is not necessary as $\delta \Rightarrow a$, where a is an atomized NDD: we just need to compute conjunctions with the atoms on paths from a to terminal `true`, since atoms not in a never intersect with BDDs in δ. After that, we recursively apply the `update` for the successors of δ and a. But instead of enumerating all the edges of a for each $e_\delta \in edges(\delta)$, we just need to consider an edge e if any atom in $atoms(e)$ has a non-false conjunction with $label(e_\delta)$. Finally, we need to replace the atoms being split with new atoms on all related edges of atomized NDDs.

Details on the implementations of the above APIs can be found in Appendix B.

4.2 The Internals

Node table, also termed as unique table, is a core data structure used by a BDD library to hold BDD nodes. Since the BDD node table often dominates the memory cost of a BDD library, and most API calls need to manipulate the node table, the efficiency of node table will be the key to the performance of a BDD library. A node table is essentially a hash table with each BDD node is uniquely identified by a tuple $(var, low, high)$. In order to achieve a higher efficiency, instead of using the built-in data structures like `HashMap` in Java, modern BDD libraries like Buddy (C++), JDD (a Java version of Buddy), etc., choose to implement their own node table with a dynamic array, with customized hash algorithms and collision resolution methods.

Different from BDD, where each node has exactly two successors, an NDD node can have a variant number of successors, and thus need a variant size of memory. Due to the non-uniform NDD node size, it is nontrivial to adapt the array-based implementation of BDD node table to NDD. Fortunately, we find that the number of NDD nodes is much smaller compared to that of BDD nodes, and therefore choose a simpler implementation based on hash map, as shown in Figure 7. Specifically, we represent an edge set of a node with a hash map , where the key is the next node of the edge, and the value is the label of the edge. Then, for each NDD variable, we also maintain a hash map, where the key is an edge set,

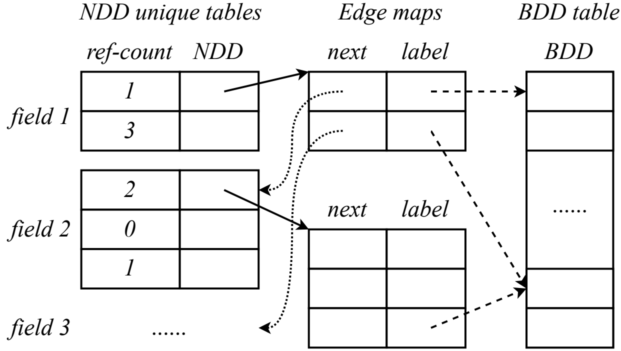


Figure 7: The node table of NDD.

and the value is an NDD node which has such an edge set. This hash map can be seen as a table holding nodes of the same NDD variable. Then, the NDD node table is just a set of these tables, one for each NDD variable. Note that we do not need *var* in the hash key, since each variable has its own sub-table. Also note that the hash function used for the edge set should be associative and commutative; otherwise, two nodes with the same successors and labels, but in a different order may not have the same hash code, and will be treated as different nodes. Our current implementation uses the Java native `HashMap` (JDK 1.8), which satisfies such requirements. Further optimizations may work better but we find the above implementation already achieves a considerable improvement for network verifiers based on BDDs.

As shown in Figure 7, for each NDD node, the library creates a hash map consisting of all its edges. Then, the library stores a tuple of a reference count, and a pointer to the hash map of its edges, into the NDD node table, which is also a hash map. Note that each field has its own NDD node table, and the BDD nodes for different fields are *logically isolated*: they still share the same BDD node table, but do not interfere with each other.

Redundancy elimination is a function that is necessary for (RO)BDDs to achieve a canonical representation of boolean formulas. Recall that (RO)BDDs use two redundancy eliminations rules *uniqueness* and *no redundant nodes (tests)*. Besides the above two rules, NDDs apply a third rule *no redundant edges*. To apply this rule, the NDD library checks whether two edges of an NDD node point to the same successor; if so, it merges them into a single edge whose condition is the disjunctions of the conditions of the two merged edges. For *uniqueness*, BDD views two nodes as the same if their two edges point to the same successor. NDD, however, requires that not only their edges point to the same set of successors, but also require that for each successor, the test conditions (BDD) are the same. For *no redundant node*, BDD views a node is redundant if both the high and low edges point to the same successor. NDD, however, applies this rule in a slightly different way: since NDD always merges edges pointing to the same successor, an NDD node is viewed as redundant if it has a single edge with condition `true`.

Garbage collection is important for BDD libraries to reduce memory usage. A BDD library maintains a reference count for each node, and marks those nodes with reference count of zero as *dead*. When the size of a node table is reaching a specified threshold, it frees all those dead BDD nodes to make more space for new nodes. Similar to a BDD library, an NDD library also uses reference counts for `gc`. One thing worth mentioning is the dependency of the `gc` processes of the NDD library and the BDD library. Firstly, since an NDD edge uses BDDs to represent the test conditions, when the NDD `gc` frees some NDD nodes and their edges, it should decrease the reference count of BDD nodes used by these edges. Secondly, we should ensure *an NDD `gc` is always be called right before a BDD `gc`.* The reason is as follows. Suppose the BDD `gc` is going to free all dead BDD nodes, some of which are still in use on NDD edges (this can happen when an NDD node is marked as dead but not freed by a `gc`). Those BDD nodes can't be freed without NDD `gc` freeing those edges, and that will affect the performance of BDD library.

Removing the `false` NDD node. In BDD, there are two terminal nodes `true` and `false`, where the `false` node and all edges pointing to it can be removed without affecting the correctness. However, most BDD libraries choose not to do so, since the benefit is marginal. In NDD, however, removing the `false` NDD node can achieve a considerable reduction in memory cost. The reason is that each edge in NDD is associated with a BDD, which represents the test condition of that edge, and by removing edges to `false`, we can save the BDDs on those edges. Moreover, since these BDDs often represent some "default" cases, they tend to be complex and freeing them often brings even larger reduction than other BDDs not on edges to `false`. For atomized NDD, we observe that the edges to the `false` NDD node often hold most of the atoms, and removing the edges allows NDD to maintain much less atoms. On our datasets, 85% edges pointing to `false` take more than 95% atoms of related field, and 30% of the edges each takes more than 2000 atoms (see Appendix A.7 for details). In addition, since the `and` operations do not care about the `false` NDD node (the conjunction of an NDD with it is still `false`), removing edges to the `false` node can reduce the running time by not enumerating these edges.

Operation cache. Similar to BDD libraries, our NDD library also maintains an operation cache, a hash table where the key is the operation, e.g., (`op`, *a*, *b*) and the value is the result of the operation. Our experiment shows an operation cache can significantly speed up NDD operations, thereby making the NDD-based verifiers faster. For example, NDD-based SRE [58] with an operation cache is shown to be over 10× faster than without an operation cache.

4.3 Using NDD

We show how to use the APIs of NDD library to realize the same data plane verification task as using BDD libraries. The

Table 2: Number of modified LOC when replacing BDD with NDD, for different verifiers. Note that the numbers exclude those for trivial string replacement (e.g., "BDD" → "NDD").

Verifiers	AP Verifier	APT	APKeep	SRE	Batfish
Library	JDD	JDD	JDD	JDD	JavaBDD
Add	+25	+84	+66	+205	+8
Delete	-159	-167	-311	-53	-7

bottom of Figure 2 shows the code snippet for data plane verification based on NDD. The logic is mostly the same with that of BDD, with key differences including: (1) NDD creates a single variable with 32 bits, instead of 32 boolean variables (Line 1); (2) NDD uses a single line of `atomize` to automatically compute atoms, after which the predicate of each port becomes an atomized NDD (Line 3); (3) When computing reachability based on atomized NDDs, verifiers use the same `and` operation as the standard (non-atomized) NDD, and therefore fully agnostic of atoms (Line 8). Details about this case, and two more use cases for incremental data plane verification and data plane verification with transformers can be found in Appendix F.

5 Implementation

The NDD Library. We implemented a library of NDD with ∼2K lines of Java code, based on JDD [8], a BDD library used by many verifiers like AP Verifier [51], APT [53], APKeep [57], SRE [58], etc. In addition, we integrated the NDD library as a new factory in JavaBDD, with another ∼100 lines of code, so that verifiers using JavaBDD, e.g., Batfish [4], Campion [47], can switch to use NDD instead.

Network verifiers based on NDD. We use the NDD library to replace the BDD library in 3 data plane verifiers, and 2 control plane verifiers. Table 2 shows the number of modified LOC for each verifier. As we can see, the number of added LOC is small since the NDD exposes similar APIs as BDD, and existing verifiers can directly use them without modifications. On the other hand, the number of deleted LOC is slightly larger. The reason is that NDD natively supports the computation of atoms, and therefore we remove the corresponding codes in those verifiers. An exception is SRE, which uses customized algorithms to traverse BDDs, and we need to adapt those algorithms to NDDs, with more modified LOC. Therefore, we conclude that the NDD library offers a drop-in replacement for BDD-based verifiers, with a small modifications to their original code base. More details can be found in Appendix A.1.

6 Evaluation

We evaluate the performance of NDD in terms of running time and memory cost, and compare to that of using BDD.

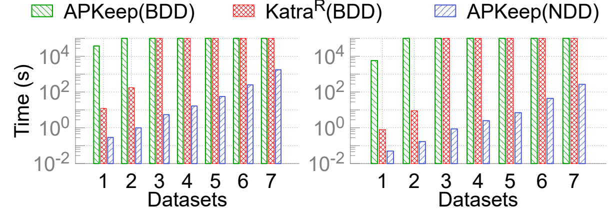


Figure 8: Running time in datacenter networks.

6.1 Setup

Datasets. We use three types of datasets for experiments.

(1) Virtualized datacenter networks (multi-layer networks). We use the virtualized datacenter network datasets (based on VXLAN) to evaluate the performance of NDD on packet transformers. The datasets are synthesized according to production networks [36,55], consisting of 7 different sizes (from 6 to 500 leaf routers), and for each size there are 5 types of updates, e.g., adding a subnet or virtual network. Details can be found in both [36] and Appendix A.2.

(2) WAN and campus networks (single-layer networks). We use three real networks that are extensively used by data plane verifiers [30,33,51,53,57], i.e., the Stanford network [5] of 16 nodes, the Internet2 network [6] of 9 nodes, and the Purdue network [46] of 1646 nodes. The Internet2 has only forwarding rules, while the other two have both forwarding and ACL rules. Details can be found in Appendix A.2.

(3) Fat trees. To evaluate the performance of our NDD libraries for control plane verification, we use different sizes (from 20 to 500 nodes) of fat trees running BGP. The datasets are synthesized and used by SRE [58].

BDD variable ordering. When comparing with data plane verifiers, we use the ordering of `srcIP < dstIP < srcPort < dstPort < protocol` for single-layer networks, and `innerDstIP < srcIP < vni < srcEPG < dstEPG < outerDstIP` for multi-layer networks. When comparing with SRE, we follow the same ordering as SRE, i.e., placing header variables before link variables, and the header variables follow the above ordering as the single-layer networks. For Batfish, we use its default ordering of `dstIP < srcIP < dstPort < srcPort < protocol < flags`.

All experiments run on a server with 2× 12-core Intel Xeon CPUs @ 2.3GHz and 256G RAM (a single core is used).

6.2 Data plane verification: running time

We run the two data plane verifiers, i.e., APKeep and Katra^R, to check all-pair reachability on the multi-layer (virtualized datacenter) networks and the single-layer networks, and compare the running time when using BDD and NDD.

Multi-layer networks (snapshot). Figure 8(a) shows the total time of computing atoms and checking reachability on all snapshots of the virtualized datacenter networks. APKeep(NDD) is the only verifier that runs to complete on

Table 3: Performance in virtualized datacenter networks. Columns 2-4 show memory overhead, columns 5-7 show BDD nodes used for representing atoms, columns 8-10 show the number of atoms and columns 11-13 show the number of new atoms introduced by updates. TO means timeout ($> 24h$).

# Leaf nodes	Memory overhead(GB)			BDD nodes			Atoms			New atoms		
	APKeep (BDD)	Katra^R (BDD)	**APKeep (NDD)**	APKeep (BDD)	Katra^R (BDD)	**APKeep (NDD)**	APKeep (BDD)	Katra^R (BDD)	**APKeep (NDD)**	APKeep (BDD)	Katra^R (BDD)	**APKeep (NDD)**
6	4.36	0.17	**0.01**	1354365	114536	**2195**	28077	5467	**112**	8043	1309	**7**
10	> 20	2.88	**0.01**	TO	513069	**3716**	TO	25542	**195**	TO	4062	**10**
20	> 32	> 115	**0.05**	TO	TO	**8659**	TO	TO	**483**	TO	TO	**14**
50	> 80	> 115	**0.16**	TO	TO	**19775**	TO	TO	**1173**	TO	TO	**14**
100	> 180	> 115	**0.45**	TO	TO	**37127**	TO	TO	**2321**	TO	TO	**15**
200	> 180	> 115	**1.42**	TO	TO	**69934**	TO	TO	**4623**	TO	TO	**15**
500	> 180	> 115	**7.19**	TO	TO	**161603**	TO	TO	**11544**	TO	TO	**15**

all snapshots, while the other two BDD-based verifiers either time out (>24h), or run out of memory (>256GB). For snapshots that all verifiers finish, APKeep(NDD) is much faster than the other two: for dataset 1 with 6 leaf nodes, APKeep(NDD) runs >10× and $10^5\times$ faster than Katra^R(BDD) and APKeep(BDD), respectively.

Multi-layer networks (update). Figure 8(b) shows the average time to incrementally check properties after the 5 types of update. Since incrementally checking the updates relies on finishing the check of snapshots, we set the running time of the two BDD-based verifiers to maximum for snapshots that they can not finish. Still, APKeep(NDD) is orders-of-magnitude faster than two BDD-based verifiers.

Single-layer networks (snapshot). Figure 9 shows the running time for the three single-layer networks. We do not include the results for Katra^R, since it has no performance improvement over APKeep on these single-layer networks. We can see that for Stanford and Internet2, the running times of BDD and NDD are comparable, while for Purdue, using NDD is 3× faster than using BDD. The reason for the limited speedup is that these networks mostly match a single field, i.e., dstIP (even Stanford and Purdue have ACL rules that match 5-tuples, the number of ACL rules is quite small). This makes the benefits of decoupling fields less significant than in multi-layer networks.

Single-layer networks (update). To simulate updates on single-layer networks, we choose the Purdue dataset, which has relatively more ACL rules such that the result are more statistically significant. For each ACL rule r, we initially insert all ACL rules other than r, and then insert r and measure the time to update the network model. As shown in Figure 10, the running time for APKeep(BDD) is highly skewed: most updates cost less than 50ms, while 1% of them cost 300ms to 2000ms. In contrast, when using NDD, the update time is always below 150ms. A possible reason is that NDD creates much less atoms after each update: for 7 out of 2707 updates, APKeep(BDD) creates >3000 new atoms (some of them will be merged later), while APKeep(NDD) just creates at most 24 new atoms after each update (see Appendix A.4 for details).

6.3 Data plane verification: memory cost

Table 3 shows the memory usage of different verifiers, computed as the difference of JVM memory usage before and after verification. As shown in Table 3 columns 2-4, the memory usage of APKeep(NDD) is orders-of-magnitude smaller than its BDD counterpart, and also Katra^R(BDD). Specifically, for the network of 6 leaf nodes, the memory cost of APKeep(NDD) is less than $1/100$ and $1/10$ that of APKeep(BDD) and Katra^R(BDD), respectively. The reason that APKeep(BDD) requires 4.36GB memory here is that the network matches 6 fields, leading to an explosion of atoms (28077 atoms creates over 10^6 BDD nodes). As shown in Columns 5-7, the number of BDD nodes of the two BDD-based verifiers is quite large, quickly blowing up the BDD node table (a maximum of $O(2^{32}/3)$ nodes as in JDD), while the number of BDD nodes for APKeep(NDD) is relatively small. This shows the power of NDD in reducing the number of BDD nodes. As for single layer-networks, using NDD reduces the memory cost of APKeep by half on the Purdue dataset, and has comparable memory cost on the Stanford and Internet2 datasets, compared to using BDD.

A major factor for the memory cost is the number of atoms. We continue to study the the number of atoms using NDD and BDD, respectively. Columns 8-10 reports the number of atoms, where the number for APKeep(NDD) is the sum of atoms for each field. As we can see, the number of atoms for APKeep(NDD) increases almost linearly with network size, and is orders-of-magnitude smaller than those of the other two BDD-based verifiers. Columns 11-13 further shows the number of newly created atoms after each update. We can see that BDD-based verifiers may create thousands of atoms after updates, while APKeep(NDD) always creates a rather small number (≤15) of atoms after each update. The above results show NDD allow verifier scale better to large networks without explosion of atoms.

6.4 Handling packet transformer

To understand how NDD supports packet transformers, we synthesize a set of NAT rules for the Purdue dataset. We

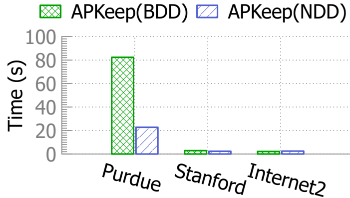


Figure 9: Running time of APKeep on single-layer networks.

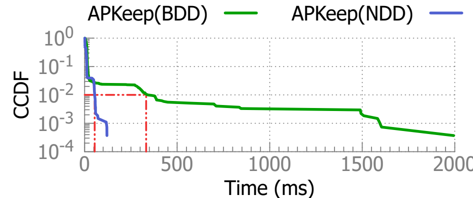


Figure 10: The complement CDF of time for APKeep to update one ACL rule on Purdue dataset.

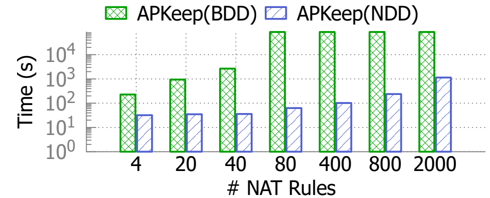


Figure 11: Running time of APKeep on Purdue dataset with NAT rules.

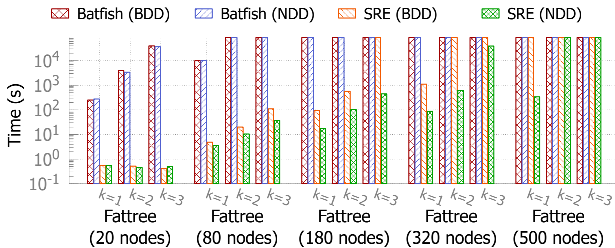


Figure 12: Running time of Batfish and SRE on fattrees.

randomly select k ports not connected to any other, and for each of them add two NAT rules matching srcIP or dstIP , one for each direction. Then, we randomly select another k ports and for each of them add two twice NAT rules (matching srcIP and dstIP simultaneously), one for each direction.

Figure 11 shows the running time of APKeep when using BDD and NDD, respectively. We can see that APKeep(NDD) is 10× faster than APKeep(BDD) even when there are only 4 NAT rules; and 100× faster when 40 NAT rules are inserted. APKeep(BDD) runs out of memory after inserting 80 NAT rules, while APKeep(NDD) runs to complete up to 2000 NAT rules. The reason that NDD can handle packet transformers more efficiently than BDD is NDD computes atoms separately for srcIP and dstIP, leading to much fewer atoms.

6.5 Control plane verification

We run two control plane verifiers, i.e., Batfish and SRE, using both NDD and BDD, to check all-pair reachability on fat tree topologies under $k = 1,2,3$ link failures. For Batfish(NDD), we create 4 NDD variables for the 32-bit IP address, each of which has 8 bits. For SRE(NDD), we create 8 NDD variables, each with the same number of bits representing the up/down state of links. An exception is the 500-node fattree, for which we create 16 NDD variables.

As shown in Figure 12, SRE has a better scalability by replacing BDD with NDD: for $k = 1$ failure on the 500-node fattree, SRE(BDD) aborts due to BDD table overflow, while SRE(NDD) can run to complete. We confirm the improved scalability is owing to the reduction of BDD nodes (Appendix A.5). For Batfish, however, the improvement of scalability is limited by using NDD. This is because the datasets only use a single field, i.e., dstIP, and Batfish doesn't have link variables as SRE. While we still find the number of BDD nodes in Batfish is reduced by using NDD (Appendix A.5). The reason is the synthesized IPs in this dataset share the same 16-bit prefix, i.e., 10.0.∗.∗. Using BDD, all packet sets need a copy of BDD nodes for those 16 bits (these nodes can not be shared by the packet sets since they differ in the last 16 bits).

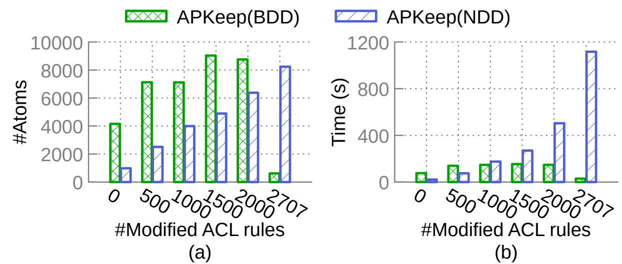


Figure 13: NDD vs. BDD on modified Purdue dataset, with different number of ACL rules matching all the 5 fields.

6.6 The importance of field locality

To study the importance of field locality for the effectiveness of NDD, we modified ACL rules in the Purdue dataset, so that more rules will match all the five fields (against randomly generated values). As shown in Figure 13, both the number of atoms and the running time of APKeep(NDD) grow linearly with the number of modified rules. In contrast, the number and time of APKeep(BDD) significantly decrease when all ACL rules match all fields. The reason is that when all rules match all fields, the phenomenon of field locality disappears, so the number of atoms when using BDD is not so large. This implies that the performance gain of NDD relies on the assumption of field locality, which generally hold in networks, but can theoretically fail (as in this synthesized scenario).

7 Related Work

Besides the classic BDD, researchers have proposed many variants of Decision Diagrams.

Decision diagrams with multiple terminals. Multi Terminal BDD (MTBDD) [12, 22], also termed Algebraic Decision Diagram (ADD), is a variant of BDD but with more than two terminals. MTBDD can model multiple outcomes (rather than true/false) of decisions, and have been used by some network

verifiers [16, 28, 35]. However, since MTBDDs still branch at a single bit, they acknowledge the same limitations of BDD when applied to network verification.

Decision diagrams branching on constraints. Forwarding Decision Diagram (FDD^b) [42, 43] is proposed to compile NetKat programs (a set of `IF-THEN-ELSE` clauses). Each FDD^b node has two edges representing whether a packet satisfies a *specific* constraint, e.g., $dstIP = 10.0.0.1$, or not. When using it for network verification, we may need many levels of nodes for each field, e.g., another level for $dstIP \in 192.168.0.0/16$. Constrained Decision Diagram (CDD) [21] is proposed to represent all solutions to a constraint satisfaction problem (CSP). Different from FDD^b, a CDD node can have multiple edges, each representing a SMT constraint, e.g., $x_1 < x_2 + 4$. In this sense, CDD represents a set of SMT constraints in the form of a decision diagram, but without solving them. To obtain a truth assignment, CDD still needs to try all possible values for variables in the constraints of each node (`cddFindall` algorithm [21]). This can be inefficient when the domain of the variable is large, e.g., if we encode `dstIP` with single variable, CDD needs to try 2^{32} values for each edge labeled by a constraint on the variable of `dstIP`.

Decision diagrams branching on a field. Multi-valued Decision Diagram (MDD) is a diagram that branches on multiple bits each time [44]. However, MDD has an edge for each single value of these bit. This which makes MDD not scalable for network verification, since a field often has many bits (e.g., a 32-bit IPv4 address requires 2^{32} edges). Similarly, Interval Decision Diagram (IDD) [45] and Firewall Decision Diagram (FDD^a) [29] branch based on one or multiple ranges of a field. However, network verifiers often need to reason about arbitrary sets of packets, which lead to a huge number of disjoint ranges, and using IDD or FDD^a is not scalable. NDD differs from the above DDs by using BDDs for a relatively small number of edges for each field.

Other implementations of BDD. Apart from JDD and JavaBDD, there are other excellent implementations of BDD. For example, Sylvan [49] is a BDD implementation which supports parallelism in both operation and algorithm level. Decision Diagram [1] is a native .NET implementation of BDD with better performance than many other BDD libraries. It is used by ZEN [17], a general and compositional framework for network modeling. However, the above implementations still suffer the memory/computation inefficiency when applied to network verification, as discussed in this paper.

Other data structures. Early verifiers like Veriflow [34] and HSA [33] use Trie and ternary bit vectors (TBV), respectively, to represent packet sets. However, such data structures cannot compactly encode arbitrary disjoint ranges. BDD-based verifiers like AP Verifier [52] and APKeep [57] have been shown to be orders of magnitude faster than HSA (based on TBV) and Veriflow (based on Trie). More recently, researchers proposed ddNF [18] and #PEC [31], which uses disjunction normal form to represent packet sets. All these data structures may perform well in some specific settings, but are still unable to supersede BDDs in general scenarios.

In sum, existing data structures are either not compact or inefficient for multiple fields: BDD is compact but the bit-level encoding makes it inefficient for multiple fields; other structures like Trie, CDD, MDD, etc. are more efficient for multiple fields by encoding at field level, but they encode each field with integers, ranges, constraints, etc., are thus not compact. In addition, when applied to network verification, all these data structures require verifiers to implement their own computation of ECs over the whole network state, resulting in a large number of ECs.

Theoretically, NDD can be seen as a special form of symbolic automata [23], where the transitions among nodes are boolean formulas. This paper is not fundamentally contributing to such theory, but provides a new design of design diagram library customized for network verification tasks.

8 Discussion

Extending NDD with multiple terminals. Besides the standard BDD, the Multi-Terminal BDD (MTBDD) [12, 22] has also been used by some network verifiers, e.g., ShapeShifter [16], YU [35], ProbNV [28], etc. For example, YU uses MTBDD to model the different (multiple) traffic volumes on a link, under different failures, so as to verify whether the link will be overloaded under some failures. We tentatively think NDD can be extended to support multiple terminals, such that it can apply to a broader range of network verifiers.

How to partition fields? For network verification, partitioning the packet header into to multiple fields, and encoding each field with an NDD variable achieves good performance. For more general scenarios where fields may not be quite clear (links as in SRE), we leverage some heuristics: (1) variables encoding a complete semantics together should be partitioned into the same field, (2) variables appearing in the same operand of BDD operations should be partitioned into the same field.

9 Conclusion

This paper introduced Network Decision Diagram, a new decision diagram customized for network verification. We designed and implemented a library for NDD, and show it can significantly reduce the memory and time for five BDD-based network verifiers. Our future work includes extending NDD with multiple terminals, making NDD library distributed and parallel, and testing the NDD library with more verifiers.

Acknowledgement. We thank our shepherd Ryan Beckett, and all the anonymous NSDI reviewers for their valuable comments and suggestions. This work is supported by the National Natural Science Foundation of China (No. 62272382). Peng Zhang is the corresponding author of this paper.
This work does not raise any ethical issues.

References

[1] A native .NET implementation of BDDs. https://github.com/microsoft/DecisionDiagrams.

[2] AP Verifier. https://www.cs.utexas.edu/users/lam/NRL/Atomic_Predicates_Verifiers.html.

[3] APKeep. https://github.com/XJTU-NetVerify/apkeep.

[4] Batfish. https://github.com/batfish/batfish/.

[5] Hassel, the header space library. https://bitbucket.org/peymank/hassel-public.

[6] Internet2. http://www.internet2.edu.

[7] JavaBDD, a Java BDD library. https://javabdd.sourceforge.net/.

[8] JDD, a pure Java BDD and Z-BDD library. https://bitbucket.org/vahidi/jdd/.

[9] A. Abhashkumar, A. Gember-Jacobson, and A. Akella. Tiramisu: Fast multilayer network verification. In *USENIX NSDI*, 2020.

[10] E. Al-Shaer, W. Marrero, A. El-Atawy, and K. Elbadawi. Network configuration in a box: Towards end-to-end verification of network reachability and security. In *IEEE ICNP*, 2009.

[11] H. R. Andersen. An introduction to binary decision diagrams. *Lecture notes, available online, IT University of Copenhagen*, page 5, 1997.

[12] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebric decision diagrams and their applications. *Formal methods in system design*, 10:171–206, 1997.

[13] R. Beckett and A. Gupta. Katra: Realtime verification for multilayer networks. In *USENIX NSDI*, 2022.

[14] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. A general approach to network configuration verification. In *ACM SIGCOMM*, 2017.

[15] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. Control plane compression. In *ACM SIGCOMM*, 2018.

[16] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. Abstract interpretation of distributed network control planes. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–27, 2019.

[17] R. Beckett and R. Mahajan. A general framework for compositional network modeling. In *ACM Workshop on Hot Topics in Networks*, 2020.

[18] N. Bjørner, G. Juniwal, R. Mahajan, S. A. Seshia, and G. Varghese. ddnf: An efficient data structure for header spaces. In *Haifa Verification Conference*, 2016.

[19] M. Brown, A. Fogel, D. Halperin, V. Heorhiadi, R. Mahajan, and T. Millstein. Lessons from the evolution of the batfish configuration analysis tool. In *ACM SIGCOMM*, 2023.

[20] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 100(8):677–691, 1986.

[21] K. C. Cheng and R. H. Yap. Constrained decision diagrams. In *AAAI*, volume 5, pages 366–371, 2005.

[22] E. M. Clarke, K. L. McMillan, X. Zhao, M. Fujita, and J. Yang. Spectral transforms for large boolean functions with applications to technology mapping. In *International Design Automation Conference*, 1993.

[23] S. Drews and L. D'Antoni. Learning symbolic automata. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2017.

[24] S. K. Fayaz, T. Sharma, A. Fogel, R. Mahajan, T. Millstein, V. Sekar, and G. Varghese. Efficient network reachability analysis using a succinct control plane representation. In *USENIX OSDI*, 2016.

[25] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *USENIX NSDI*, 2015.

[26] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan. Fast control plane analysis using an abstract representation. In *ACM SIGCOMM*, 2016.

[27] N. Giannarakis, D. Loehr, R. Beckett, and D. Walker. NV: an intermediate language for verification of network control planes. In *ACM PLDI*, 2020.

[28] N. Giannarakis, A. Silva, and D. Walker. ProbNV: probabilistic verification of network control planes. *Proceedings of the ACM on Programming Languages*, 5(ICFP):1–30, 2021.

[29] M. G. Gouda and X.-Y. Liu. Firewall design: Consistency, completeness, and compactness. In *IEEE ICDCS*, 2004.

[30] A. Horn, A. Kheradmand, and M. R. Prasad. Delta-net: Real-time network verification using atoms. In *USENIX NSDI*, 2017.

[31] A. Horn, A. Kheradmand, and M. R. Prasad. A precise and expressive lattice-theoretical framework for efficient network verification. In *IEEE ICNP*, 2019.

[32] S. K. R. Kakarla, A. Tang, R. Beckett, K. Jayaraman, T. Millstein, Y. Tamir, and G. Varghese. Finding network misconfigurations by automatic template inference. In *USENIX NSDI*, 2020.

[33] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *USENIX NSDI*, 2012.

[34] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey. VeriFlow: Verifying network-wide invariants in real time. In *USENIX NSDI*, 2013.

[35] R. Li, Y. Yuan, F. Ye, M. Liu, R. Yang, Y. Yu, T. Guo, Q. Ma, X. Zeng, C. Xu, et al. A general and efficient approach to verifying traffic load properties under arbitrary k failures. In *ACM SIGCOMM*, 2024.

[36] X. Liu, P. Zhang, H. Li, and W. Sun. Modular data plane verification for compositional networks. *Proceedings of the ACM on Networking*, 1(CoNEXT3):1–22, 2023.

[37] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese. Checking beliefs in dynamic networks. In *USENIX NSDI*, 2015.

[38] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright. Virtual extensible local area network (vxlan): A framework for overlaying virtualized layer 2 networks over layer 3 networks. Technical report, 2014.

[39] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. Godfrey, and S. T. King. Debugging the data plane with Anteater. In *ACM SIGCOMM*, 2011.

[40] C. Perkins. IP encapsulation within IP. Technical report, 1996.

[41] S. Prabhu, K. Y. Chou, A. Kheradmand, B. Godfrey, and M. Caesar. Plankton: Scalable network configuration verification through model checking. In *USENIX NSDI*, 2020.

[42] S. Smolka, S. Eliopoulos, N. Foster, and A. Guha. A fast compiler for netkat. In *ACM SIGPLAN International Conference on Functional Programming*, 2015.

[43] S. Smolka, P. Kumar, D. M. Kahn, N. Foster, J. Hsu, D. Kozen, and A. Silva. Scalable verification of probabilistic networks. In *ACM PLDI*, 2019.

[44] A. Srinivasan, T. Ham, S. Malik, and R. K. Brayton. Algorithms for discrete function manipulation. In *IEEE international conference on computer-aided design*, 1990.

[45] K. Strehl and L. Thiele. Interval diagrams for efficient symbolic verification of process networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(8):939–956, 2000.

[46] Y.-W. E. Sung, S. G. Rao, G. G. Xie, and D. A. Maltz. Towards systematic design of enterprise networks. In *ACM CoNEXT*, pages 1–12, 2008.

[47] A. Tang, S. K. R. Kakarla, R. Beckett, E. Zhai, M. Brown, T. Millstein, Y. Tamir, and G. Varghese. Campion: debugging router configuration differences. In *ACM SIGCOMM*, 2021.

[48] B. Tian, X. Zhang, E. Zhai, H. H. Liu, Q. Ye, C. Wang, X. Wu, Z. Ji, Y. Sang, M. Zhang, et al. Safely and automatically updating in-network acl configurations with intent language. In *ACM SIGCOMM*. 2019.

[49] T. Van Dijk and J. Van de Pol. Sylvan: multi-core framework for decision diagrams. *International Journal on Software Tools for Technology Transfer*, 19:675–696, 2017.

[50] D. Wang, P. Zhang, and A. Gember-Jacobson. Expresso: Comprehensively reasoning about external routes using symbolic simulation. In *ACM SIGCOMM*, 2024.

[51] H. Yang and S. S. Lam. Real-time verification of network properties using atomic predicates. In *IEEE ICNP*, 2013.

[52] H. Yang and S. S. Lam. Real-time verification of network properties using atomic predicates. *IEEE/ACM Transactions on Networking*, 24(2):887–900, 2015.

[53] H. Yang and S. S. Lam. Scalable verification of networks with packet transformers using atomic predicates. *IEEE/ACM Transactions on Networking*, 25(5):2900–2915, 2017.

[54] F. Ye, D. Yu, E. Zhai, H. H. Liu, B. Tian, Q. Ye, C. Wang, X. Wu, T. Guo, C. Jin, et al. Accuracy, scalability, coverage: A practical configuration verifier on a global WAN. In *ACM SIGCOMM*, 2020.

[55] L. You, J. Zhang, Y. Jin, H. Tang, and X. Li. Fast configuration change impact analysis for network overlay data center networks. *IEEE/ACM Transactions on Networking*, 30(1):423–436, 2021.

[56] P. Zhang, A. Gember-Jacobson, Y. Zuo, Y. Huang, X. Liu, and H. Li. Differential network analysis. In *USENIX NSDI*, 2022.

[57] P. Zhang, X. Liu, H. Yang, N. Kang, Z. Gu, and H. Li. APKeep: Realtime verification for real networks. In *USENIX NSDI*, 2020.

[58] P. Zhang, D. Wang, and A. Gember-Jacobson. Symbolic router execution. In *ACM SIGCOMM*, 2022.

*The appendices are subject to changes, and the latest version can be found at https://xjtu-netverify.github.io/papers/2025-ndd-a-decision-diagram-for-network-verification.

A More Evaluation Results

A.1 Implementation Details

In the following, we give more details on the implementations of the 5 verifiers with the NDD library.

AP Verifier [51] and APT [53]. AP Verifier is a pioneering work that pre-compute atomic predicates based on BDD, and APT extends AP Verifier to support packet transformers. We used their open-source codes [2], and modified them to use the NDD library. The modifications are to remove their implementations of computing atomic predicates, and re-implement it with the `atomize` API of NDD.

APKeep [57]. APKeep is a data plane verifier that extends AP Verifier to incrementally update atomic predicates, also based on BDD. We used its source codes [3], and modified it to use the NDD library. The modifications are to remove its previous implementation of incremental update of atomic predicates, and re-implement it with the `update` API of NDD.

Katra [13]. Katra is a data plane verifier proposed for multi-layer networks. Katra introduces a novel *partial equivalence classes* to significantly reduce the total number of ECs, and is proven to be more efficient than APKeep when verifying multi-layer networks. We are interested in whether we can achieve the same or even better efficiency as Katra, by simply changing the underlying data structure, without computing partial equivalence classes. Therefore, we re-implemented Katra based on BDD, referred to as Katra^R, and further extended it to support VXLAN overlay networks in Table 5. We confirmed the correctness of Katra^R based on the datasets synthesized according to [13]. However, we did not implement the NDD version of Katra since the core difference of Katra from APKeep is its novel *partial equivalence classes* based on BDD, and replacing BDD with NDD will make Katra no different from APKeep (NDD).

SRE [58]. SRE is a control plane verifier that jointly encodes failure scenarios and packet headers, so as to scale to large networks. We used the authors' implementation and replaced its JDD library with our NDD library. For packet headers, we encode each header field with an NDD variable; for failure scenarios, we divide the set of all links evenly into 8 groups, and encode each link group with an NDD variable.

Batfish [19, 25]. Batfish is an open-source control plane verifier, which uses JavaBDD [7]. We replaced the BDD factory in JavaBDD with our NDD factory, with small modifications of Batfish, including declaring fields and initializing the unique table and operation cache.

A.2 Statistics of datasets

Table 4 and Table 5 show the experimental datasets.

Table 4: Datasets for WAN and campus networks.

Networks	Nodes	Links	Forwarding rules	ACL rules
Purdue	2,159	3,607	3.52×10^6	2707
Stanford	124	182	3.84×10^3	686
Internet2	9	56	1.26×10^5	0

Table 5: Datasets for virtualized datacenter networks.

Networks	Leafs (VPCs)	Subnets (VRFs)	Forwarding rules	MCS rules	Static routes
1	6	36	856	12	12
2	10	100	2,508	20	20
3	20	400	13,428	40	40
4	50	1,000	59,808	100	100
5	100	2,000	144,508	200	200
6	200	4,000	388,908	400	400
7	500	10,000	1,528,538	1,000	1000

A.3 Micro-benchmark results

Table 6: Running time of each stage and the number of atoms for forwarding and ACL rules on Purdue dataset.

Verifiers	IPv4		ACL		Property
	Time(s)	Atoms	Time(s)	Atoms	Check(s)
APKeep(BDD)	13.88	265	22.37	3893	46.11
APKeep(NDD)	14.87	265	7.04	411	0.88

The per-stage running time for Purdue is shown in Table 6. NDD uses more time than BDD when processing forwarding rules (Column 2), due to the overhead of maintaining per-field atoms; while NDD uses roughly $1/3$ the time of BDD when processing ACL rules (Column 4), since the ACL rules match 5-tuples. For checking properties, NDD takes much less time than BDD (Column 6). This is because using BDD will result in much more atoms than using NDD, as shown in column 5: by using NDD, APKeep can reduce the number of atoms by around 89.4%.

A.4 Number of new atoms after each update

Figure 14 shows the complement cumulative distribution of the number of new atoms for each update in Purdue dataset.

A.5 Number of BDD nodes for Control Plane Simulation

Figure 15 shows the number of used BDD nodes for SRE and Batfish on the fattree datasets. The number of BDD nodes of Batfish(BDD) is many times more than that of Batfish(NDD), and the number of BDD nodes of SRE(BDD) is more than that of SRE(NDD) by orders of magnitude, which shows memory inefficiency of BDD.

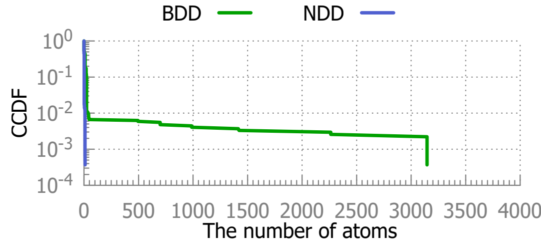


Figure 14: The complement cumulative distribution of the number of new atoms for each update in Purdue dataset.

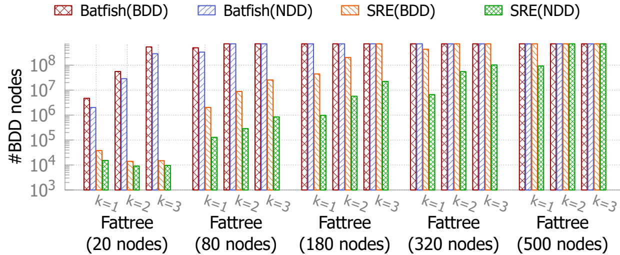


Figure 15: The number of used BDD nodes for control plane simulation on fattree networks.

A.6 Number of edges in NDD

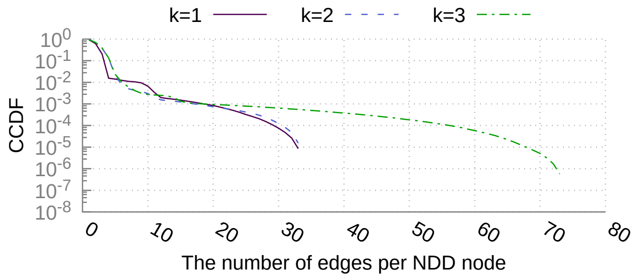


Figure 16: The complement cumulative distribution of the number of edges for each NDD node.

Figure 16 shows the number of edges for each NDD node created by SRE(NDD) on the 80-nodes fattree network. More than 99% of the nodes have no more than 10 edges, and all the nodes have no more than 75 edges, which shows that NDD can efficiently merge redundant edges with a high compression ratio.

A.7 Number of atoms on edges to `false`

Figure 17 (a) shows the number of atoms that are labeled on each edge pointing to the terminal node `false`. We can see that almost 30% of these edges have more than 2000 atoms. Figure 17 (b) shows the percentage of atoms on edges pointing to the `false` node to the total number of atoms of the corresponding field. We can see that for 85% of these edges have more than 95% of atoms of the corresponding field.

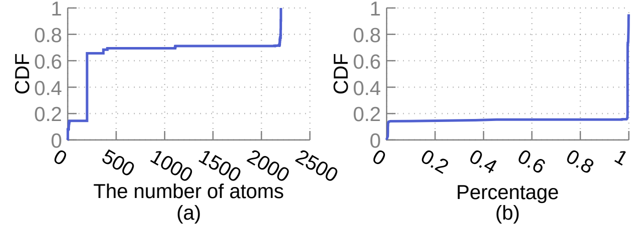


Figure 17: (a) the number and (b) the percentage of atoms on edges pointing to `false`.

Table 7: Time to solve the NQueens problem.

N	6	7	8	9	10	11	12
BDD (s)	0.017	0.023	0.04	0.223	0.615	2.567	19.109
NDD (s)	0.012	0.019	0.038	0.176	0.344	2.257	12.417

A.8 Performance of NDD for general problems

We compare the performance of BDD and NDD in solving the NQueens problem, a widely recognized benchmark for BDD libraries. For BDD, we use the implementation of the JDD library, which is basically an implementation of algorithms in [11]. For NDD, we declare each row as a field (therefore there are N fields), and implement the same algorithm. Table 7 shows that the time of BDD is around $1.5\times$ that of NDD for $N = 12$. This is because the BDD library has an considerable overhead for aligning variables when computing BDD conjunctions, especially for large N. This indicates the phenomenon of "field locality" may not be limited to network verification, and NDD may be used to speedup other applications.

A.9 BDD variable orderings

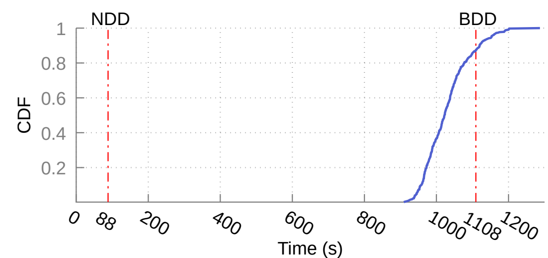


Figure 18: Running time of SRE on fattree (320 nodes, $k = 1$) with random variable orderings. The two dotted lines represent the running time of SRE(BDD) and SRE(NDD) reported in Figure 12, respectively.

As briefly mentioned in §2.2, one may wonder whether we can mitigate limitations of BDD with a better variable ordering. We discuss this for two different types of verifiers.

First, for verifiers that compute atoms, e.g., AP Verifier, APT and APKeep, the performance is highly related to the number of atoms, while little affected by the variable ordering. Therefore, variable ordering makes little differences to their

performance when suffering explosion of atoms.

Second, for verifiers that do not compute atoms, e.g., SRE, variable ordering indeed impact the performance. Here, we use SRE(BDD) as an example to study such impact. We found an ordering with header variables (`dstIP`) before link variables, which was used by SRE, always leads to better performance. Therefore, when we run SRE(BDD), we first declare the variables of `dstIP`, and then declare the link variables with random order. Figure 18 reports the running time, where we can see the time varies from 800s to 1200s with different orderings, but is still much larger than that of SRE(NDD). This confirms that re-ordering the variables does not fundamentally make BDD more efficient for network verification.

A.10 Comparing NDD to BDD arrays

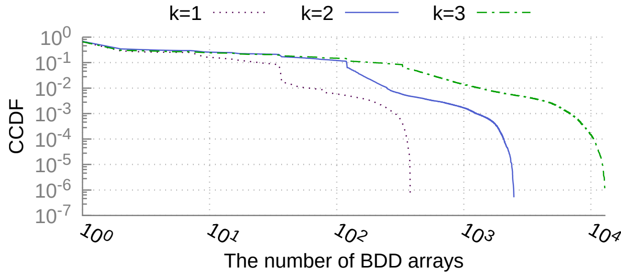


Figure 19: The complement cumulative distribution of the number of BDD arrays to represent a predicate in SRE.

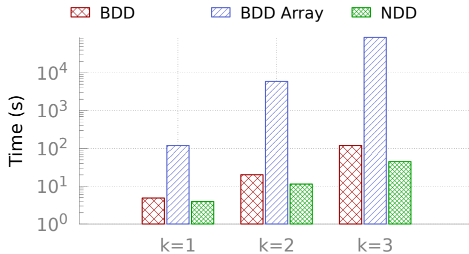


Figure 20: Running time on 80-node fattree of verifiers based on three methods.

We implement SRE based on BDD arrays, which simply assembles per-field BDDs by conjunctions, and compare its performance to that based on NDD and BDD, on a 80-node fattree. Figure 19 shows the number of BDD arrays representing each predicate, which is as large as $10^2, 10^3, 10^4$ for $k = 1,2,3$, respectively. Figure 20 further shows the running time. We can see that using BDD arrays is orders-of-magnitude slower than using NDDs, and even slower than directly using BDDs. Especially when $k = 3$, using BDD arrays times out due to too many BDD arrays.

B Algorithms of NDD operations

Algorithm 1 shows the implementation of `apply`(op, a, b), where `op` can be `and`, `or`, `diff`; a and b are two NDDs. For trivial cases such as either a or b is a terminal node, or $a = b$, the algorithm directly returns the result (Lines 1-2). If the variable of a and b is the same (Line 3), the algorithm recursively performs `apply` for each successor of a and each successor of b (Lines 4-5). Note that, the algorithm merges redundant edges in Line 9. If the variable of a has a higher order, the algorithm recursively performs `apply` for b and each successor of a (Lines 10-15). Finally, it looks for an NDD node (or create a new node) from node table representing the result and returns it (Line 16). The algorithm for atomized NDDs is the same with the above, except that BDD operations of *and* and *or* in Line 6, 9 and 15 change to set operations of *intersect* and *union*. Note that, when we enable the optimization of removing the `false` node (§4.2), the implementations of `or` and `diff` need some minor modifications.

Algorithm 1: `apply`$(op,\ a,\ b)$

Input: a logical operator $op \in \{and, or, diff\}$, and two NDDs a and b.
Output: the result of logical operation $a\ op\ b$.

```
1  if a or b is a terminal node, or a = b then
2      return handleTrivialCases(op, a, b);
3  if var(a) = var(b) then
4      foreach e_a ∈ edges(a) do
5          foreach e_b ∈ edges(b) do
6              label ← bdd.and(label(e_a), label(e_b));
7              if label ≠ bdd.false then
8                  next ← apply(op, next(e_a), next(e_b));
9                  edgeMap[next] ←
                       bdd.or(edgeMap[next], label);
10 else
11     if var(a) > var(b) then
12         swap(a, b);
13     foreach e_a ∈ edges(a) do
14         next ← apply(op, next(e_a), b);
15         edgeMap[next] ← bdd.or(edgeMap[next],
               label(e_a));
16 return mk(var(a), edgeMap);
```

Algorithm 2 shows the implementation of `exist`. If the field variable of a has lower priority than $field$, the algorithm directly returns a as the result (Lines 1-2). If the field variable of a is equal to $field$, the algorithm applies existential quantification by merging all successors of a through NDD's function of `or` (Lines 3-7). If the field variable of a has higher priority than $field$, the algorithm invokes a new recursion for each successor of a and merges results (Lines 8-12).

Algorithm 3 shows the implementation of `atomize`. It takes

Algorithm 2: `exist`(a, $field$)

Input: an NDD a and a variable $field$.
Output: an NDD with existential quantification on a for variable $field$.

```
1  if var(a) > field then
2      return a;
3  else if var(a) = field then
4      sum ← ndd.false;
5      foreach e ∈ edges(a) do
6          sum ← apply(or, sum, next(e));
7      return sum;
8  else
9      foreach e ∈ edges(a) do
10         next ← exist(next(e), field);
11         edgeMap[next] ← bdd.or(edgeMap[next],
             label(e));
12     return mk(var(a), edgeMap);
```

Algorithm 3: `atomize`(N)

Input: a set of NDDs N.
Output: a set of atomized NDDs N' that are logically equivalent to N.

```
1  labels ← allLabels(N);
2  foreach field ∈ allFields(N) do
3      atoms[field] ← computeAtoms(labels[field]);
4  N' ← {};
5  foreach n ∈ N do
6      N' ← N' ∪ atomizeNDD(n, atoms);
7  return N';
```

a set of NDDs N to be atomized as input and returns a set of atomized NDDs N', which are logically equivalent to N. The algorithm first collects all labels (BDDs) used by some of the input NDDs, and group these labels by field (Line 1). Then, the algorithm computes atoms for each field (Lines 2-3), in a similar way as AP Verifier does. After that, it transforms the each input NDD into a logically-equivalent atomized NDD (Lines 4-6).

Algorithm 4 shows the implementation of `update`. It takes as input an NDD δ and an atomized NDD a, δ is a predicate to be atomized and a is used to accelerate the function. *split* in Line 2 is a map that maps each old atom to be split to 2 new atoms. The algorithm first recursively traverses δ and a, and detects the atoms to be split (Line 3), then performs the split by replacing each old atom in labels of existing atomized NDDs with 2 new atoms in *split* (Line 4). Since an atom may be split many times by different label BDDs in δ, the function repeatedly performs steps above until *split* is empty after Line 3. If the δ and a have the same variable, for each edge e_δ of δ and each edge e_a of a, the function intersects the BDD on e_δ with only atoms held by e_a to detect atoms to be split (Lines 9-20). Note that, it only traverses successors of edges whose atoms intersect with the label (Lines 19-20). If the variable of δ has higher priority than that of a, the algorithm intersects the labels with each atom of $var(\delta)$ (Lines 21-27).

Algorithm 4: `update`(δ, a)

Input: an NDD δ to be atomized; an atomized NDD a satisfying $\delta \Rightarrow a$.

```
1  do
2      split ← {};
3      updateRec(δ, a, split);
4      splitAtoms(split);
5  while ¬split.isEmpty();
6  Function updateRec(δ, a, split):
7      if δ = ndd.true then
8          return;
9      else if var(δ) = var(a) then
10         foreach e_δ ∈ edges(δ) do
11             foreach e_a ∈ edges(a) do
12                 hit ← false;
13                 foreach ap ∈ label(e_a) do
14                     t ← bdd.and(ap, label(e_δ));
15                     if t ≠ bdd.false then
16                         hit ← true;
17                         if t ≠ ap then
18                             split[ap] ←
                                 {t, bdd.diff(ap, t)};
19                 if hit then
20                     updateRec(next(e_δ), next(e_a), split);
21     else if var(δ) < var(a) then
22         foreach e ∈ edges(δ) do
23             foreach ap ∈ atoms[var(δ)] do
24                 t ← bdd.and(ap, label(e));
25                 if t ≠ false and t ≠ ap then
26                     split[ap] ← {t, bdd.diff(ap, t)};
27             updateRec(next(e), a, split);
```

Figure 21 shows an example for `update`, where (b) is an NDD δ to be atomized, and (c) is an atomized NDD a. Labels on δ are per-field BDDs and labels on a are per-field atoms. Relations of per-field BDDs and atoms are shown in (a), where $A_1 = a_1 \vee a_2'$ means A_1 contains (is implied by) a_1 and partially intersects (has a non-false conjunction) with a_2. The operation first chooses the left edge of δ and a, computes conjunctions of A_1 with a_1 and a_2, respectively, and splits a_2 into new atoms a_2' and $\neg a_2'$, such that A_1 can be atomized. Then, the operation tries to match the edges labeled by B_1 and b_1 in the second field. After that, the operation tries to match the left edge of δ and the middle and right edges of a, respectively, finds that A_1 doesn't intersect with a_3 or a_4, and skips calculation in the second field. As shown above, A_1

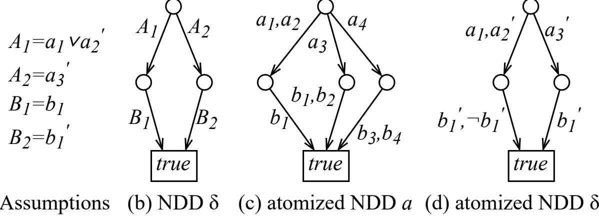


Figure 21: An example for `update` operation. A_i and B_i are BDDs; a_i and b_i are atoms. $a_i^{'}$ means it implies a_i, i.e., $a_i^{'} \Rightarrow a_i$.

only needs to intersect with atoms a_{1-4} which appear on the paths from a to terminal `true`, rather than all atoms of the field (e.g., atoms on the edge of a pointing to terminal `false`, which are not shown in the figure), thereby accelerating the operation. The steps are similar for the right edge of δ. Finally, atoms a_2, a_3 and b_1 are split, such that δ can be atomized as shown in (d).

C Proof of canonicity of NDD

Definition 4. ***Boolean formulas.*** *A boolean formula F has $n_1 + n_2 + \cdots + n_k$ boolean variables, which can be grouped into k fields $f_1, f_2, \ldots, f_k$, where field f_i has n_i consecutive boolean variables $x_{s_i+1}, x_{s_i+2}, \ldots, x_{s_i+n_i}$, where $s_i = n_1 + n_2 + \cdots + n_{i-1}$.*

Definition 5. ***Truth assignments and restrictions.*** *A truth assignment of a field f_i is denoted as $B_i = [b_{s_i+1}, b_{s_i+2}, \ldots, b_{s_i+n_i}]$, where $b_j \in \{true, false\}$. A restriction of F with respect to a variable x and a boolean constant b, denoted as $F[b/x]$, is defined as the same formula F, except that the variable x is replaced with b. Similarly, we can extend the definition to a field f_i and a truth assignment B_i, as*

$$F[B_i/f_i] = F[b_{s_i+1}/x_{s_i+1}, b_{s_i+2}/x_{s_i+2}, \ldots, b_{s_i+n_i}/x_{s_i+n_i}]$$

Suppose bdd is a BDD defined over f_i, and $\forall B_i \in bdd$, $F[B_i/f_i]$ is the same, then define the restriction of F with respect to field f_i and BDD bdd as $F_{bdd} = F[B_i/f_i], \forall B_i \in bdd$.

Shannon expansion is the theoretical basis of BDD: a BDD node encoding F branches on x, with a *high* edge assigning `true` to x and pointing to the successor encoding $F[1/x]$, and a *low* edge assigning `false` to x and pointing to the successor encoding $F[0/x]$.

Definition 6. ***Single-bit Shannon expansion.*** *First define the if-then-else operator as $x \to y_0, y_1 = (x \wedge y_0) \vee (\neg x \wedge y_1)$. Then, the Shannon expansion of formula F with respect to a boolean variable x is defined as:*

$$F = x \to F[1/x], F[0/x].$$

Since each NDD variable (i.e., field) has multiple bits, we extend the above single-bit Shannon expansion to the following multi-bit Shannon expansion.

Definition 7. ***Multi-bit Shannon expansion.*** *Let $\mathcal{BDD}$ be a set of BDDs satisfying $\bigvee_{bdd \in \mathcal{BDD}} bdd = true$, and $\forall bdd_1, bdd_2 \in \mathcal{BDD}$, $bdd_1 \neq bdd_2 \Rightarrow bdd_1 \wedge bdd_2 = false$. Then, the Shannon expansion on formula F with respect to field f_i is:*

$$F = \bigvee_{bdd \in \mathcal{BDD}} (bdd \wedge F_{bdd}) = f \to F_{bdd_1}, F_{bdd_2}, \ldots, F_{bdd_n}.$$

Definition 8. ***The formula encoded by an NDD.*** *Suppose u is an NDD node having a set of edges $e_1, e_2, \ldots, e_n$, Then, u encodes a boolean formula F^u defined as:*

$$F^u = var(u) \to F_{label(e_1)}, F_{label(e_2)}, \ldots, F_{label(e_n)}. \quad (2)$$

Lemma 2. ***Canonicity of NDD.*** *For any boolean function $F(x_1, x_2, \cdots, x_N)$ over N boolean variables with ordering $x_1 < x_2 < \cdots < x_N$, there is exactly one RONDD u with k NDD variables and ordering $f_1 < f_2 < \cdots < f_k$, where f_i has n_i bits $x_i^1 < x_i^2 < \cdots < x_i^{n_i}$, and $N = \sum_{i=1}^k n_i$, such that u can represent F.*

Proof. Clearly the lemma holds for $k = 0$, and for $k \geq 1$, we prove it by induction on k. The proof is adapted from that for BDD canonicity [11].

Assume now that we have proven the lemma for k fields $(f_2, f_3, \cdots, f_{k+1})$. We prove the lemma for $k+1$ fields $(f_1, f_2, \cdots, f_{k+1})$. For a boolean function F with $k+1$ fields, the following two steps proves (1) there exists an NDD node u that can represent F, and (2) if there is another NDD node v, such that $F^v = F$, then we must have $v = u$.

(1) The existence of an NDD for boolean formula F. For f_1 with n_1 bits, there are 2^{n_1} distinct truth assignments, represented as a set $\mathcal{B}$. Each $B \in \mathcal{B}$, the restriction $F[B/f_1]$ has no more than k bits, and thus there exists an NDD node that can represent it (according to our assumption for k). We can partition $\mathcal{B}$ into m classes $C_1, C_2, \ldots, C_m$, satisfying that $\forall B_1, B_2 \in \mathcal{B}, F[B_1/f_1] = F[B_2/f_1] \Leftrightarrow \exists C_i, B_1 \in C_i, B_2 \in C_i$. For each C_i, there exists an NDD node v_{C_i} such that $\forall B \in C_i$, v_{C_i} represents each $F[B/f_1]$, and a BDD bdd_{C_i} that can represent $\bigvee_{B \in C_i} B$. We create an NDD node u with $var(u) = f_1$ and m edges, satisfying that $\forall e_i \in edges(u) \Rightarrow label(e_i) = bdd_{C_i}, next(e_i) = v_{C_i}$. According to Definitions 7 and 8, we have

$$F = f_1 \to F_{bdd_{C_1}}, F_{bdd_{C_2}}, \ldots, F_{bdd_{C_m}} = F^u$$

, meaning node u represents the formula F.

(2) The uniqueness of NDD for boolean formula F. There are two cases to consider.

(i) u has a single edge e with $label(e) = true$ and $next(e) = v_{C_1}$. This violates RONDD's condition of **no redundant node**. By reducing redundant nodes, u degenerates to v_{C_1}, which is canonic (canonicity of NDD with $\leq k$ fields).

(ii) u has more than one edges. Assume v is an NDD node with $F^v = F$, and we need to show $u = v$. First, we must have $var(v) = f_1$; otherwise if $var(v) \neq f_1$, then the

assignment of f_1 doesn't affect F, and u must be an NDD node with a single edge, which violates the assumption that u is a RONDD. For $\forall e_u \in edges(u)$ and a truth assignment $\forall B_1 \in label(e_u)$, we can find an edge $e_v \in edges(v)$ such that $B_1 \in label(e_v)$, because v must encode any truth assignment in f_1. In addition, we have $next(e_u) = next(e_v)$ because these two nodes encode the same formula $F[B_1/f_1]$ (canonicity of NDD with $\leq k$ fields). For any other truth assignment $\forall B_2 \in label(e_u), B_2 \neq B_1$, we can also find an edge $e_{v'} \in edges(v)$ such that $B_2 \in label(e_{v'})$. We can get that $next(e_v) = next(e_u) = next(e_{v'})$, and if $e_v \neq e_{v'}$, it violates RONDD's condition of **no redundant edges**. Therefore, we get that $e_v = e_{v'}$. That means, for $\forall e_u \in edges(u)$, there must exist $e_v \in edges(v)$, such that $label(e_u) \Rightarrow label(e_v)$. Vice versa, we get $label(e_v) \Rightarrow label(e_u)$ and therefore $label(e_u) = label(e_v)$ and $next(e_u) = next(e_v)$. Therefore, v and u have the same variable and set of edges. According to the **uniqueness** of RONDD, it follows that $v = u$.

□

D An example for explosion of atoms

We use an example in Figure 22 to show why the number of atoms can explode when using BDDs. For simplicity of illustration, we consider only `dst IP/Port` fields, and suppose ACL rules at port 1 of D have not been inserted.

Using BDD, a data plane verifier, i.e., APKeep, works in following steps. First, it splits each node (device) in (a) into *elements* in (b), each of which represents a single action in networks, e.g., forwarding, filter. Then, the verifier encodes match conditions of rules with BDDs, and computes port predicates for each element. The predicates are shown in (c) with labels of $B_1 - B_6$. Packets encoded in a port predicate will be forwarded to the related port or pass the related filter. After that, the verifier computes atoms in (d), each of which represents a set of packets with the same global behaviour. Suppose for now there are only forwarding rules matching `dstIP`. There are only 3 different behaviors, which correspond to $dstIP \in$ 00, 01 and 1*, respectively. Suppose there are also ACL rules matching `dstPort`, and the number of behaviours (atoms) will increase to 3 × 3 = 9, a cross-product of the 3 different behaviors for each of the two fields (B_{1-3} for `dstIP`, $B_4, B_5 - B_4$ and $\neg B_5$ for `dstPort`). As shown in (d), the three sub-structures of BDD nodes for `dstIP` are combined with the three sub-structures of BDD nodes for `dstPort`, resulting in 20 nodes in total. As shown in (e), the verifier replaces each port predicate in (b) with an equivalent set of atoms in (d). Then, as shown in (f), the verifier injects all atoms into A and traverses the network with a set of reachable atoms. At each element, it intersects the set of reachable atoms with atoms of each port of the element by using set operation $\cap$ and $\cup$, and forwards the result to the next element. Finally, atoms of $A_{1-3,5,6,8}$ can pass port 1 of D from A.

Using NDD, in contrast, the data plane verifier also computes port predicates with logical operations of $\wedge$, $\vee$ and $\neg$ as shown in (g), whose edges are labeled by BDDs in (c). Then, it collects all per-field BDDs on the edges, and calculates per-field atoms. As shown in (h), comparing with $3 \times 3 = 9$ atoms for the BDD-based verifier, there are only $3 + 3 = 6$ atoms for the NDD-based verifier, labeled by D_{1-3} and E_{1-3}. After that, as shown in (j), the verifier atomizes each port predicate of NDDs in (g) by atomized NDDs in (i). For example, the BDD label B_5 on the edge of C_5 encoding $dstPort = 0*$, is replaced by atoms E_1 and E_2 encoding $dstPort = 00$ and 01, respectively. Finally, the verifier traverses the network and computes reachability by the operation $\wedge$ and $\vee$ of atomized NDD and the reachable packets are encoded by F_8.

E An example for BDD arrays

To show the limitation of simply assembling per-field BDDs as arrays, let us take a look at an example in Figure 23 (a), where we use BDD arrays to represent packet sets that are reachable from A to D in the network in Figure 3. Specifically, each array has four BDDs representing four fields, i.e., `src IP/Port` and `dst IP/Port`. Initially, an array of {****, **, **, **} which represents all packets is injected at device A. Then, each device filters the arrays based on its forwarding or ACL rules, and sends the resultant arrays to the next hops. At device D, we end up with 9 arrays which collectively represent all reachable packets from A to D. These 9 arrays correspond to a product of 3×3 paths from the root nodes of the three BDDs in Figure 4 (a) to the BDD terminal node `true` (3 sub-paths for $srcIP/Port$ and 3 sub-paths for $dstIP/Port$). 9 arrays are not a big issue in the example, but what we want to highlight here is the number of arrays can grow quickly due to a cross-product effect. In real scenarios, there can be over $O(10^4)$ arrays even when the network is very small (see Appendix A.10 for details). Figure 23 (b) shows the NDD representing reachable packets from A to D on the right. NDD compactly represents compositions of sub-paths from different fields by a form of decision diagram. Specifically, the 3 sub-paths for $dst\ IP/Port$ are represented once by an NDD node of the $dstIP$ field, and the 3 sub-paths for $src\ IP/Port$ share such NDD node without representing the 3 sub-paths for $dst\ IP/Port$ by 3 times, and therefore occupies much less memory than BDD arrays in practice.

F Use cases of NDD in network verification

F.1 Automatic atomization

We use AP Verifier [51], a data plane verifier, as an example to show how to use the NDD APIs to automatically compute atoms.

Step 1. Computing port predicates. Using NDD, we first use the `createVar` function to declare the header fields (i.e.,

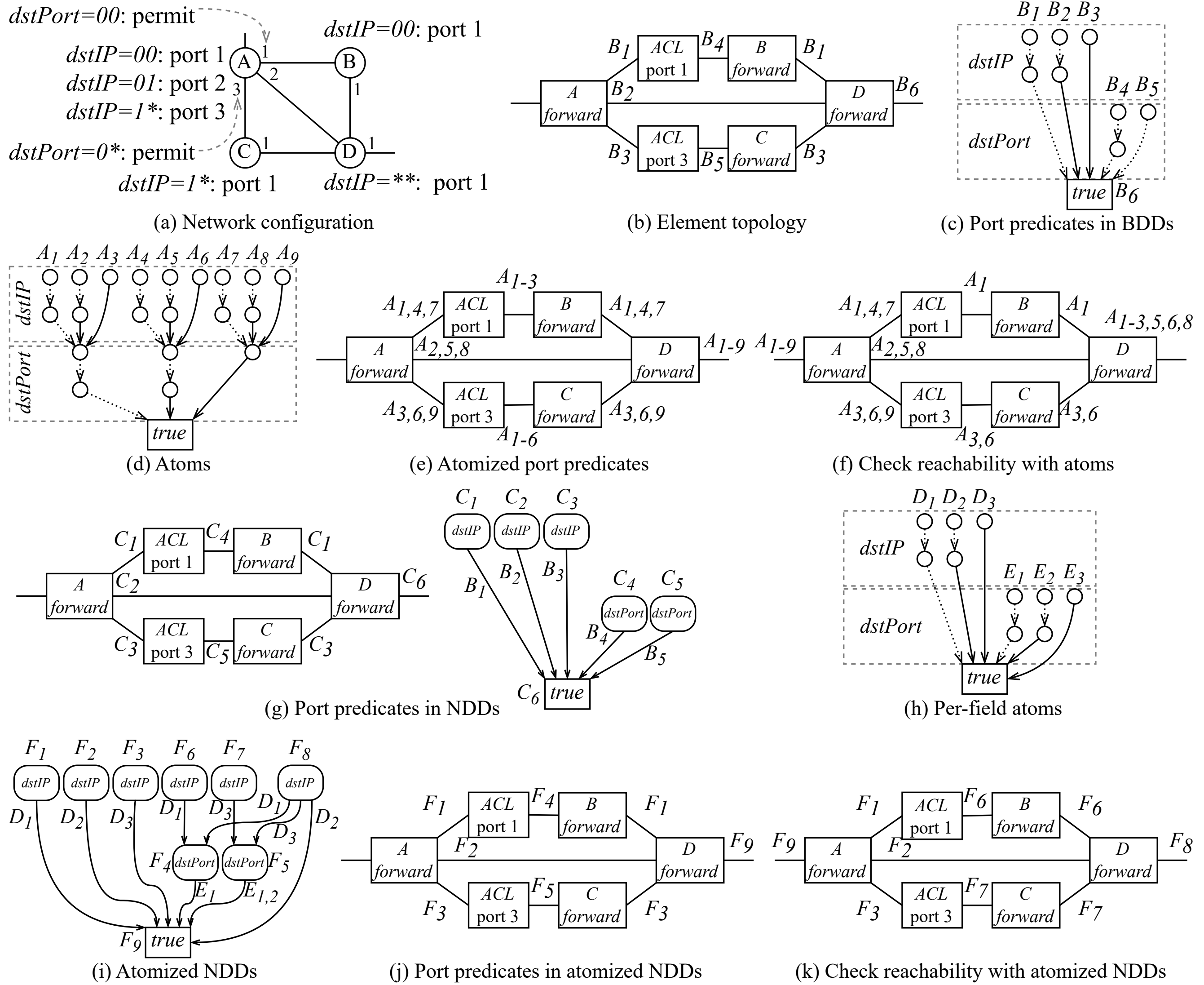


Figure 22: An example for explosion of atoms when using BDDs.

5-tuple). Then, we use the same algorithms as in AP Verifier to compute port predicates, with the only difference that BDD operations are replaced with NDDs operations (`apply`).

Step 2. Computing atomic predicates. Instead of computing atoms using the port predicates with a custom algorithm (a two-fold loop implemented in AP Verifier), we use the `atomize` function with the port predicates as input. The function will compute atoms for each field, and return the *atomized NDDs*, a new NDD with each BDD on an edge replaced by the equivalent sets of atoms.

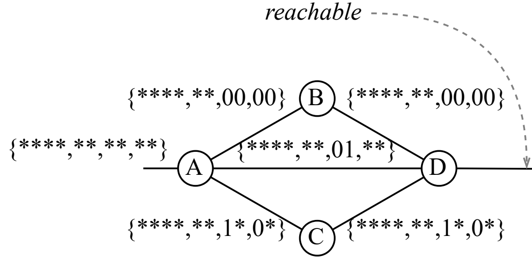
Step 3. Checking reachability. We follow the same approach in AP Verifier to check network reachability, which maintains a set of atoms during the traversal, and performs intersections on the set with those of the ports being traversed. The difference is that the traversal uses logical operations on atomized NDD (e.g., `and`, `or`), instead of set operations, as in AP Verifier. In this sense, the process is similar to using BDDs without computing atoms, but can still enjoy a speedup due to computing per-field atoms. Note here the logical operations on atomized NDDs are different from those of standard NDDs: since labels of edges are sets of atoms in atomized NDD, the operations on labels become set operations instead of BDD operations.

F.2 Incremental update of atoms

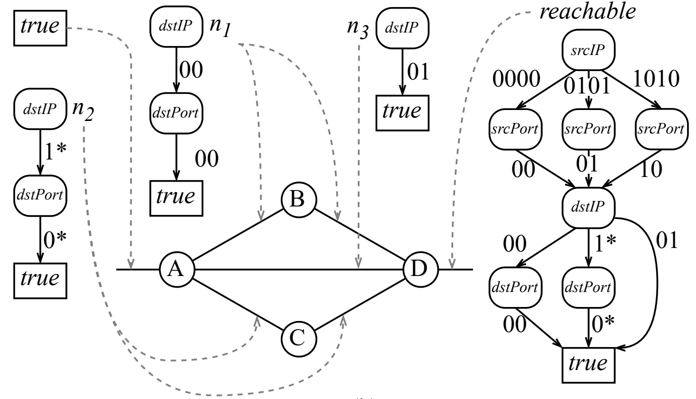
We use APKeep [57], an incremental data plane verifier, as an example to show how to use the NDD APIs to incrementally update the atoms.

Step 1. Identifying changes. This step takes rule insertions or deletions as input, and calculates changes of forwarding behaviors. A change is represented as a tuple $(from, to, \delta)$, meaning that packets with headers in δ are forwarded to port $from$ before the update and to port to after the update. This

A→B→D: {0000,00,00,00}{0101,01,00,00}{1010,10,00,00}
A→D: {0000,00,01,**}{0101,01,01,**}{1010,10,01,**}
A→C→D: {0000,00,1*,0*}{0101,01,1*,0*}{1010,10,1*,0*}



(a) Arrays of per-field BDDs



(b) NDDs

Figure 23: Computing reachable packets from A to D in the example in Figure 3, (a) and (b) use arrays of per-field BDDs and NDDs, respectively. Arrays/NDD on each link represent the reachable packets at that position, each element of an array represents a per-field BDD with an ordering of $\{srcIP, srcPort.dstIP, dstPort\}$.

step is identical to that of APKeep.

Step 2. Updating atoms. This step takes changes from step (1) and updates the set of atoms. For each change in the form of $(from, to, \delta)$, it first gets the atomized port predicate a of port $from$ and invokes `update`(δ, a) to update atoms.

Step 3. Transferring atoms. After step (2), δ can be atomized. This step atomizes δ and transfers it from port $from$ to port to using the `diff` and `or` operations for atomized NDD.

F.3 Handling packet transformers

We show how to use the NDD APIs to efficiently handle packet transformers, which refer to rules that *rewrite*, *encapsulate*, or *decapsulate* packets. A packet transformer can be represented as a tuple $(match, field, action)$, where $match$ is the matching condition, $field$ is the field to be transformed, and $action$ is the value to rewrite/encapsulate.

Step 1. Computing port predicates. For each action of transformers, e.g., encapsulating a new IP header with destination IP 10.0.0.1/32 for packets from source IP 192.0.1.0/24, the verifier creates a logical port on the device of the transformer. Each logical port P_i has two predicates: $P_i.match$, an NDD encoding the match conditions, e.g., `innerSrcIP`= 192.0.1.0/24, and $P_i.action$, an NDD encoding the actions, e.g., `outerDstIP`= 10.0.0.1/32. All these predicates are computed as in AP Verifier [51].

Step 2. Atomizing predicates. In this step, the verifier atomizes the predicates computed by step 1, using the NDD API `atomize`, in the same way as §F.1. The differences is that both the $P_i.match$ and $P_i.action$ are used for computing the atoms.

Step 3. Checking properties. In this step, when the packet set pkt (an atomized NDD) reaches a transformer, the verifier enumerates each logical port of the transformer: for each port P_i, it computes the conjunction of pkt and $P_i.match$, an atomized NDD encoding the match conditions. This is the same as §F.1. If the conjunction is not `false`, and the type of the transformer is `decapsulate` or `rewrite`, the verifier erases the original value of pkt on the field $P_i.field$, e.g., `innerSrcIP`, using the `exist` API of NDD. If the type is `encapsulate` or `rewrite`, the verifier sets a new value of pkt on the field $P_i.field$, e.g., `outerDstIP`, by computing the conjunction of pkt and $P_i.action$. The output is denoted as pkt_i. Finally, the verifier merges the output pkt_i of each logical port P_i into a packet set pkt', and forwards it to the next hop.

G The usage of BDD in network verification

Table 8 shows an incomplete list of network verifiers based on BDDs.

Table 8: Network verifiers based on BDD. ✓ means the verifier has already been re-implemented with our NDD library.

	Verifier	BDD library	Symbolic state	Impl.
Data plane	AP Verifier [51]	JDD	packets	✓
	APT [53]	JDD	packets	✓
	APKeep [57]	JDD	packets	✓
	Katra [13]	DD [1]	packets	
	PPV [55]	JavaBDD	packets	
	MNV [36]	JDD	packets	
	ConfigChecker [10]	Buddy	packets	
Control plane	Batfish [19,25]	JavaBDD	packets	✓
	SRE [58]	JDD	failures, packets	✓
	Expresso [50]	JDD	route adv., packets	
	NV [27]	CUDD	routes	
	ProbNV [28]	CUDD	routes	
	ShapeShifter [16]	unknown	routes	
	DNA [56]	JDD	packets	
	ERA [24]	JDD	routes	
	Bonsai [15]	JavaBDD	packet/route filters	
	Campion [47]	JavaBDD	packet/route filters	