UC Berkeley
UC Berkeley Previously Published Works

Title
ddNF: An Efficient Data Structure for Header Spaces

Permalink
https://escholarship.org/uc/item/6t14m6g7

ISBN
978-3-319-49051-9

Authors
Bjørner, Nikolaj
Juniwal, Garvit
Mahajan, Ratul
et al.

Publication Date
2016

DOI
10.1007/978-3-319-49052-6_4

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

ddNF: An Efficient Data Structure for Header Spaces

Nikolaj Bjørner[1], Garvit Juniwal[2], Ratul Mahajan[1], Sanjit A. Seshia[2⋆], and George Varghese[3]

[1] Microsoft Research
[2] University of California, Berkeley
[3] University of California, Los Angeles

Abstract. Network Verification is emerging as a critical enabler to manage large complex networks. In order to scale to data-center networks found in Microsoft Azure we developed a new data structure called ddNF, *disjoint difference Normal Form*, that serves as an efficient container for a small set of equivalence classes over header spaces. Our experiments show that ddNFs outperform representations proposed in previous work, in particular representations based on BDDs, and is especially suited for *incremental* verification. The advantage is observed empirically; in the worst case ddNFs are exponentially inferior than using BDDs to represent equivalence classes. We analyze main characteristics of ddNFs to explain the advantages we are observing.

1 Introduction

Just as design rule checkers statically verify hardware circuits and type checkers flag type violations in a program before execution, the emerging field of network verification seeks to proactively catch network bugs before they occur in practice by reading router tables and configuration files and checking for properties such as reachability, isolation, and loops. When compared to hardware design automation and software analysis, formal tooling around networks, is at an infant state. Networks are commonly managed using tools developed by network vendors using proprietary formats. Bare bones network tools, such as traceroute, may be the only and best option for debugging networks. Modern large scale public cloud services crave more powerful tools, including static analysis tools that can answer reachability properties in large networks.

This challenge has been recognized relatively recently: The seminal work of Xie [16] focused on reachability in IP networks and Anteater [10] provided a more abstract framework using a SAT solver to compute reachability bugs, and Header Space Analysis (HSA) [7] used a compact representation to compute all reachable headers. Later, Veriflow [8] and NetPlumber [6] found a way to do faster, incremental verification, and Network Optimized Datalog [9] implemented efficient header space verification in an expressive Datalog framework, thereby allowing higher level properties called beliefs [9] to be expressed. Properties

⋆ UC Berkeley authors supported in part by the NSF ExCAPE project (CCF-1139138).

verified include more complex path predicates (e.g., traffic between two hosts flows through a middlebox) and differential reachability (e.g., is reachability same in all load balanced paths).

Yang and Lam [17] made a crucial observation that most headers are treated the same when analyzing any given network. It is therefore much more efficient to find the relatively small set of equivalence classes of headers and then perform reachability queries based on these classes instead of integrating header computation while checking reachability. Yang and Lam base their equivalence class computation on BDDs, which succinctly represent sets of headers. Each equivalence class is a BDD (covering a disjoint set). Whenever inserting a new set, their algorithm requires examining all previous sets and performing BDD operations. While elegant and easy to implement, the overall quadratic number of BDD calls and the fact that BDDs require an overhead per bit struck us as an over fit for the networking domain.

In this paper we introduce the ddNF (disjoint difference Normal Form) data-structure and algorithms that handles the partition of headers in a particularly efficient way. In essence, our new ddNF data structure pre-computes a compressed representation of the relevant header spaces instead of the “run-time” compression employed by say HSA [7] while answering reachability queries. This transformation turns large graphs into small tractable sizes for quantitative analyses and allows faster incremental verification than the BDD based approach used in [17].

2 An overview of ddNFs

We first provide a quick overview of ddNFs. Consider a very tiny network as an example with 3 data centers A, B, and C in Fig. 1. Assume that the set of prefixes represented by B is $0\star$ and the set of prefixes of C is $1\star$. There are two routers: the leftmost router forwards every packet to its rightmost port $p1$, and the rightmost router splits traffic to B and C via the ports $p2$ and $p3$ respectively.

To compute the reachability from data center A, regular header space methods such as HSA [7] will start with the wild-card expression $\star\star$ (which represents all packets with two bits) which flows to the second router. This set of packets "splits" into two pieces. The first piece is the packets representing $1\star$ which flow down to C. The upper piece is the set of packets covered by the rule $\star$. But since the router does longest match semantics, this rule only applies to packets that do not match $1\star$, in other words to $\star\star - 1\star$. While this is indeed $0\star$ in this simple example, Header Space methods [7] keep headers in this difference of cubes representation and (to avoid state explosion) only lazily extract solutions when a symbolic packet reaches a destination.

While lazy differencing keeps the size of the header space representation manageable, it does require an intersection of an incoming header space with each set of matching rules at a router, an expensive operation that grows with the number of bits in each header and the number of matching rules. Yang and Lam [17] suggest a different technique that we refer to as pre-computed compression that has some analogies with first computing labels for each set of headers as in MPLS [13]. The idea (shown at the bottom of Fig. 1 is to rewrite each

header expression in a matching rule as a union of disjoint header expressions (called "atomic predicates" in [17]) which are then replaced by integers.

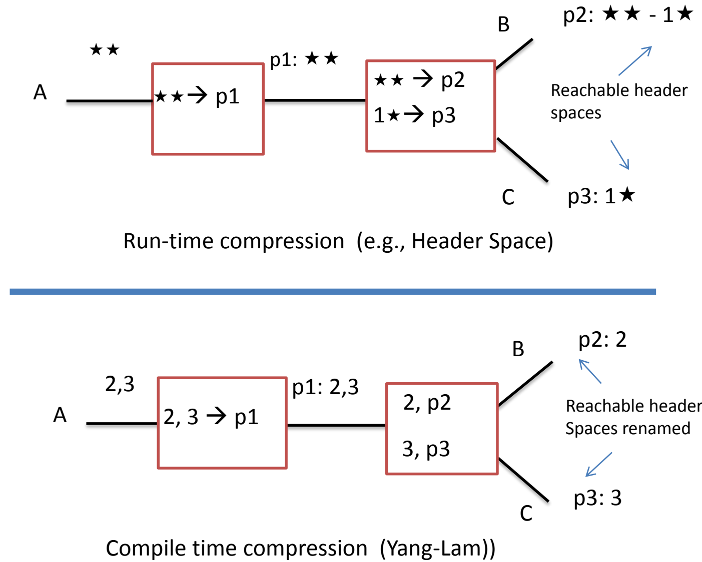

Fig. 1: Run-time vs. pre-computed Header Space Compression

2

For example, ⋆⋆ is the union of 1⋆ and 0⋆, which we represent by the integers 2 and 3 respectively (we avoided integer 1 to avoid confusion with the bit value 1) and the forwarding table is rewritten as shown. Now the same process is used to compute reachability, but this time we use lists of integers instead of wild-card matching and intersection. While it is unclear that this method works for all reachability queries, it does work very fast for basic reachability, yielding 1-2 orders of speedup compared to even the fastest run-time methods [7].

Yang and Lam [17] calculate the pre-computed set of forward equivalence classes (atomic predicates on headers) using BDDs. We use a dedicated data structure called a ddNF that we found experimentally outperforms BDDs on our benchmarks. Similar to BDDs, ddNFs are also generalized tries, but *in contrast to BDDs that branch on one bit at a time, ddNFs branch based on a subsumption relation between entire wild-card expressions.* Our approach comes with another twist: the ddNF data-structure simply indexes a partition all wild-card expressions and does not rely on aggregating these expressions for output ports as in [17].

3 Firewalls, Routers and Ternary Bit-vectors

We model an IP router as a set of rules. Firewalls are modeled as special routers that route packets either onward or to a sink that drops packets.

Each router contains a forwarding table that describes how IP packets are forwarded to another router or end-point. For example, Fig. 2 shows a snippet of a forwarding table from an Arista network switch: It says that by default

```
1         0.0.0.0/0          via 100.91.176.0, n1
2                            via 100.91.176.2, n2
3
4         10.91.114.0/25     via 100.91.176.125, n3
5                            via 100.91.176.127, n4
6                            via 100.91.176.129, n5
7                            via 100.91.176.131, n6
8         10.91.114.128/25   via 100.91.176.125, n3
9                            via 100.91.176.131, n6
10                           via 100.91.176.133, n7
11  ...
```

Fig. 2: A forwarding table snippet.

addresses are routed to either neighbor n_1 (with address 100.91.176.0) or to n_2, unless the destination address matches one of the rules below. For example, if the first 25 bits of the address match the same 25 bits of 10.91.114.0, then the packet is forwarded to either n_3, n_4, n_5 or n_6.

Ternary bit-vectors (TBVs)[4] succinctly encode matching conditions using 1, 0 and $\star$ (the latter denoting "don't care"). A TBV models a range of IP addresses by concatenating the bytes corresponding to each integer separated by dots, and then adding don't-cares for the last $32-n$ bits if the prefix is of the form A/n. Thus, 10.91.114.0/25 corresponds to the TBV 00001010 01011011 01110010 0$\star\star\star\star\star\star\star$ and 10.91.114.128/25 corresponds to the TBV 00001010 01011011 01110010 1$\star\star\star\star\star\star\star$; note that these TBVs are *incompatible* in that there is a bit position where one has a 1 while the other has a 0. We use $tbv, tbv_1, \ldots$ to denote ternary bit-vectors in $\{1, 0, \star\}^k$ of the same fixed length k. For example 10$\star\star$00 is a TBV of length $k = 6$.

Ternary bit-vectors denote a set of (concrete) bit-vectors. For example 10$\star\star$00 denotes the set $\{100000, 100100, 101000, 101100\}$. We use ternary bit-vectors and the sets they denote interchangeably. For example, we write $1 \star 0 \subset \star\star 0$.

Definition 1 (Routers). *A router, $\mathcal{R}$, is an ordered list of rules $\rho_1, \rho_2, \rho_3, \ldots, \rho_n$ where $\rho_j = \langle tbv_j, p_j \rangle$ is a pair comprising a ternary bit-vector tbv_j and an output port p_j. The rules have the following semantics: a packet header h, which is a bit-vector, matches rule $\rho_j = \langle tbv_j, p_j \rangle$ (and is forwarded to ports p_j) if each of the vectors $tbv_1, \ldots, tbv_{j-1}$ contain a conflicting bit (a 0 where h has a 1, or vice versa), whereas tbv_j has no such conflicting bit.*

The *matching condition* for rule $\rho_j = \langle tbv_j, p_j \rangle$ is the Boolean function representing the set of bit-vectors $tbv_j \backslash \{tbv_1, tbv_2, \ldots, tbv_{j-1}\}$. We denote this Boolean

[4] Also called "cubes" in the VLSI CAD literature.

function by MC_j. Note that our definition of a router does not let the router rewrite headers.

We note that router matching conditions have a special syntactic form, which we formally define below.

Definition 2 (Difference of Cubes). *A* difference of cubes *(DOC) is an expression of the form* $tbv \backslash \{tbv_1, tbv_2, \ldots, tbv_m\}$ *for ternary bit-vectors* tbv, tbv_1, tbv_2, $\ldots$, tbv_m.

For example, the DOC $1{\star}{\star}\backslash\{110, 101\}$ encodes the set $\{111, 100\}$.

A network consists of a set of connected routers. That is, a network, $\mathcal{N}w$, is a set of routers $\{\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_n\}$ along with, for each router $\mathcal{R}_i$, a map $\mathcal{L}_i$ from output ports of $\mathcal{R}_i$ to adjacent ("next hop") routers. In real networks, routers send traffic to non-routers that are end-points of traffic flow. It is a bit simpler, though, to pretend that end-points are routers that either have no incoming or no outgoing links.

A *predicate* is a Boolean function over the header bits. We adapt the definition of *atomic predicate* from Yang and Lam [17] as below.

Definition 3 (Atomic Predicates). *Given a network, a set of predicates* P_1, $\ldots$, P_n *are atomic if they are mutually disjoint, their union is equivalent to true, and in a given network* $\mathcal{N}w$*, every matching condition in any router rule for is equivalent to a union of predicates from* $\{P_1, \ldots, P_n\}$.

Note that for every set of routers there is a coarsest set of atomic predicates.

4 Pre-computed Compression via ddNFs

We wish to perform pre-computed compression by rewriting each router rule (as in [17]) using a set of integers that represent the disjoint matching conditions in order to speed up reachability checking. Instead of using BDDs to enumerate mutually disjoint matching conditions, we propose a new data structure called a ddNF that we show is more efficient for the networking domain.

As recognized in Veriflow [8] and NetPlumber [6] efficiency is important to enable *real-time incremental analysis* as router rule changes occur at high speed (for instance, to accommodate rapid virtual machine migration). For these environments, ddNFs reduce the phase of creating disjoint matching conditions from tens of seconds to a few milliseconds. Further, if rules change, but reuse existing prefixes (for example, a route change), then the ddNF requires no updates. However, in [17] because rules are aggregated on ports before computing disjoint reasons, a routing change that switches a prefixes to a new port can cause label changes.

4.1 Representing Disjoint Sets of Bit-vectors as ddNFs

Given a set of routers and rules from each router, we seek to enumerate all overlapping segments, such that each rule can be written as a set of mutually disjoint matching conditions potentially shared with other rules. For example,

if one rule matches on $10\star\star$ and a different rule matches on $1\star0\star$, then the first set is decomposed into two disjoint sets: $100\star, 10\star\star\setminus\{100\star\}$, and the second set is decomposed into $100\star, 1\star0\star\setminus\{100\star\}$. The three sets are mutually disjoint. Any member of the set $100\star$ (the members are $1001, 1000$) matches both rules, whereas members of $1\star0\star\setminus\{100\star\}$ (the members are $1101, 1100$) match only the second rule.

The *disjoint decomposed normal form*, ddNF, data structure is used to create and maintain a disjoint decomposition from DOCs. Recall that DOCs such as $100\star$, $1\star0\star\setminus\{100\star\}$ are differences of (sets of) ternary bit vectors.

Definition 4 (ddNF). *A* ddNF *is a directed acyclic graph (DAG) data structure, represented as a four-tuple*

$$\langle \mathcal{N}, E, \ell, root \rangle$$

where $\mathcal{N}$ is a set of nodes, $E \subseteq \mathcal{N} \times \mathcal{N}$ are edges, and ℓ is a labeling function mapping every node to a ternary bit-vector, and $root \in \mathcal{N}$ is a designated root node such that all nodes are reachable from it and $\ell(root) = \underbrace{\star\star\star\star}_{k}$. In addition, the ddNF data structure must satisfy the following properties:

- *Whenever $E(n, m)$ for two nodes $n, m \in \mathcal{N}$, then $\ell(m) \subset \ell(n)$.*
- *Conversely, if $n, m \in \mathcal{N}$ and $\ell(m) \subset \ell(n)$, then either $E(n, m)$ or there is a node, $m' \in \mathcal{N}$, such that $\ell(m) \subset \ell(m') \subset \ell(n)$.*
- *No two nodes are labeled by the same ternary bit-vector.*
- *The range of ℓ (i.e., the set of ternary bit-vectors labeling all nodes in $\mathcal{N}$) is closed under intersection.*

□

Figure 3 shows an example ddNF.

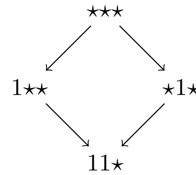

Fig. 3: Example ddNF. The root (top most node) denotes the DOC $\star\star\star\setminus\{1\star\star, \star1\star\}$, the left-most node $1\star\star\setminus\{11\star\}$, right-most node $\star1\star\setminus\{11\star\}$, and the bottom node denotes $11\star$.

The conditions for a ddNF ensure that the data structure is canonical up to isomorphism. Thus, we have

Proposition 1 (ddNFs are unique up to isomorphism). *Given a set S of ternary bit-vectors closed under intersection and containing the ternary bit-vector comprising of all $\star$ there is a unique ddNF labeled by S.*

Proof Sketch: Take the bit-vectors from S, then create the root node from the all-$\star$ bit-vector and for each of the other TBVs create an associated node. Two nodes are connected if their labels are strict subsets and there is no intermediary node labeled by a TBV that is subset-wise between them. □

We can reconstruct a DOC from a ddNF node n in the following way:

$$doc(n) = \ell(n) \backslash \{\ell(m) \mid m \in children(n)\}$$

In this way, each node represents a set disjoint from all other nodes. Conversely, we can retrieve the set of nodes that denote a difference of cube $tbv_0 \backslash \{tbv_1, \ldots, tbv_m\}$ expression by taking

$$DC(n_0) \setminus (DC(n_1) \cup \ldots \cup DC(n_m))$$

assuming the ddNF has nodes labeled $\ell(n_0) = tbv_0, \ldots, \ell(n_m) = tbv_m$, and the downward closure $DC(n)$ is defined recursively as

$$DC(n) = \{n\} \cup \bigcup \{DC(m) \mid m \in \mathcal{N}, (n, m) \in E\}$$

Note that not all nodes are necessarily representing non-empty sets. This is the case when the set of TBVs labeling children covers the TBV of the parent. Checking non-emptiness of a node amounts to checking satisfiability of the formula

$$fml(tbv) \wedge \bigwedge_i \neg fml(tbv_i)$$

where

$$fml(tbv) = \bigwedge_{i \mid tbv[i]=1} p_i \wedge \bigwedge_{i \mid tbv[i]=0} \neg p_i$$

It is however often easy to quickly determine non-emptiness in a greedy way by creating a sample bit-vector that is contained in the positive component, but different from negative components by swapping the first bit where the positive has a $\star$ and the negative has a non-$\star$ value.

4.2 Inserting into and using ddNFs

We will now describe how to update and query the ddNF data structure described in the previous section. The main operation is insertion of ternary bit-vectors. Insertion of a ternary bit-vector t can be described as follows: First of all, we insert a node n labeled by new ternary bit-vector t above the nodes closest to the *root* node that are strict subsets of t. In these positions, the new node n inherits the parents the less general node. Second, if t has a non-empty intersection with a node n', that is neither a subset or a super-set of t, then we have to create a node corresponding to $\ell(n') \cap t$ and insert this to the ddNF and ensure that t is inserted above this new node. Algorithm 1 shows pseudo-code that implements the informally described insertion algorithm. Fig. 4 and 5 show two main uses of the algorithm.

The effect of inserting ternary bit-vectors into a ddNF is characterized by the following proposition:

Algorithm 1: Insert(n,r): Insertion of node n labeled by ternary bit-vector t under a ddNF node r

Input: n - node labeled by ternary bit-vector t
Input: r - node in ddNF
Output: a node in the ddNF labeled by t

```
1  if ℓ(r) = t then
2  |  return r;
3  end if
4  inserted ← ⊥;
5  foreach (r, child) ∈ E do
6  |  if t ⊆ ℓ(child) then
7  |  |  inserted ← ⊤;
8  |  |  n ← Insert(n, child);
9  |  end if
10 end foreach
11 if inserted then
12 |  return n;
13 end if
14 foreach (r, child) ∈ E do
15 |  if ℓ(child) ⊂ t then
16 |  |  E ← {(n, child)} ∪ E \ {(r, child)}
17 |  end if
18 end foreach
19 E ← E ∪ {(r, n)};
20 foreach (r, child) ∈ E, t′ = ℓ(child) do
21 |  if t ⊈ t′ ∧ t ∩ t′ ≠ ∅ then
22 |  |  m ← fresh node labeled by t ∩ t′;
23 |  |  Insert(m, r); // Ensure child and n share m as common descendant
24 |  end if
25 end foreach
26 return n
```

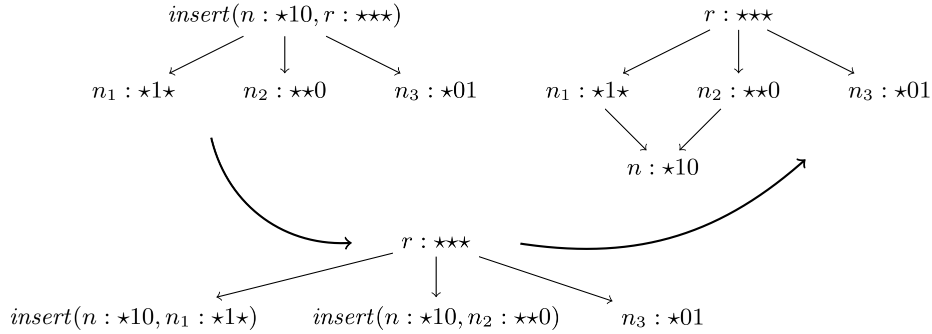

Fig. 4: Insertion below children. In the top left we insert node n labeled by $\star 10$ into a root r, which is labeled by $\star\star\star$. Both nodes n_1 and n_2 generalize n, while n_3 is disjoint from n. Insertion therefore proceeds as in the bottom of the figure by recursively inserting n into n_1 and n_2. After insertion completes, we obtain the ddNF given top right.

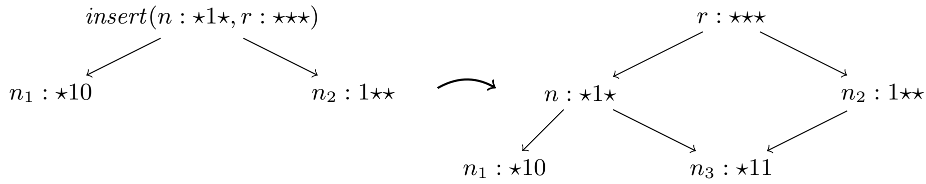

Fig. 5: Insertion with subsumption. When inserting $n : \star 1 \star$ into r we detect that n is more general than n_1, so n is inserted above n_1. On the other hand, n and n_2 are compatible, but neither generalize the other, so we create a fresh node n_3 labeled by the intersection $\star 11$ and it is inserted in a way that is illustrated in Fig. 4.

Proposition 2 (Disjoint Decomposition). *The resulting ddNF obtained after inserting the ternary bit-vectors* $tbv_1, tbv_2, \ldots, tbv_n$ *has one node corresponding to every possible distinct non-empty set obtained by intersecting some* k *of the* n *TBVs while excluding the remaining* $n - k$*.*

Another way of viewing the above result is that the ddNF has precisely one node for every disjoint region in the Venn-diagram of the sets denoted by the inserted TBVs. This property follows from the conditions in Definition 4.

Algorithm 2 shows the extraction of a ddNF from a set of routers. It also extracts a map from TBVs to labels in the extracted ddNF.

Algorithm 2: Extract a ddNF for a set of routers

Input: *Routers* a set of routers with routing rules from TBVs to ports
Output: A ddNF representing the TBVs used in *Routers*
Output: *tbv2node* a map from TBVs to labels

```
1  ddNF ← a ddNF with a single root node;
2  tbv2node ← [⋆···⋆ ↦ root];
3  foreach R ∈ Routers do
4  |  foreach ⟨tbv, p⟩ ∈ R do
5  |  |  n ← Fresh node labeled by tbv;
6  |  |  n ← Insert(n, root);
7  |  |  tbv2node[tbv] ← n;
8  |  end foreach
9  end foreach
10 return ddNF, tbv2node
```

Algorithm 3 shows how we reach our goal for pre-computed header space compression to convert each router to a small lookup table from labels to output ports. The algorithm uses the ddNF extracted from Algorithm 2. It traverses the rules, using the ddNF to extract a set of labels corresponding to each rule. It assumes that the rules are prioritized on a first-applicable basis, such that earlier rules have precedence over later rules. Thus, labels used for earlier rules cannot be used for later rules. The algorithm subtracts previously used labels

by computing $DC(tbv2node(tbv)) \setminus seen$, where *seen* are the nodes that have been used so far. To compute $DC(tbv2node(tbv)) \setminus seen$ efficiently we maintain a tag on each node. The tag is initially clear and gets set when the node is first traversed. This has the side-effect of inserting it into *seen* and also ensures that each node is traversed at most once because one can skip all nodes below an already marked node.

Algorithm 3: Convert each router R into a map R' from labels to output ports.

```
   Input: Routers a set of routers with routing rules from TBVs to ports
   Input: a ddNF for the TBVs used in Routers
   Input: a map tbv2node from TBVs to nodes in the ddNF
   Output: Routers' a set of routers whose routing rules map labels to ports
1  Routers' ← ∅;
2  foreach R ∈ Routers do
3  |  R' ← the empty map from ddNF nodes to ports;
4  |  foreach ⟨tbv, p⟩ ∈ R in order of appearance do
5  |  |  seen ← ∅
6  |  |  labels ← DC(tbv2node[tbv]) \ seen
7  |  |  foreach ℓ ∈ labels do
8  |  |  |  R'[ℓ] ← p
9  |  |  end foreach
10 |  |  seen ← seen ∪ DC(tbv2node[tbv])
11 |  end foreach
12 |  Routers' ← Routers' ∪ {R'}
13 end foreach
14 return Routers'
```

We can further optimize the labeling obtained from Algorithm 3 by using a post-processing pruning step. Define the equivalence relation $\simeq$ between two labels as follows:

$$\ell \simeq \ell' := \bigwedge_{R' \in Routers'} R'[\ell] = R'[\ell']$$

That is, two labels are equivalent if the forwarding behavior is the same for each router. We can then remove all but one equivalence class representative from each $\simeq$ class and still compute reachability. In [14], we extended this reduction by taking a transitive congruence closure. We describe this approach in more detail in Section 5.

Finally, when we check reachability for a set of headers (given by a DOC), we compute the set of labels associated with the DOC and check reachability for each of the labels.

4.3 Comparing ddNFs with BDDs

First of all let us notice that the conversion of a set of TBVs into ddNF can be exponential.

Example 1. Suppose we have a routing table with the following rules:

```
1   1.*.*        via port1
2   *.1.*        via port2
3   *.*.1        via port3
```

The rules use the ternary bit-vectors $1{\star}{\star}, {\star}1{\star}, {\star}{\star}1$. They decompose into 8 disjoint subsets and the corresponding ddNF is shown in Figure 6.

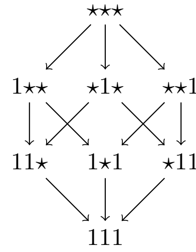

Fig. 6: Maximal ddNF

As we will later observe, the structure of real routing tables makes this worst case very unlikely; ddNFs perform very well in practice.

Yang and Lam [17] use BDDs [4] to represent header spaces and leverage BDD operations to compute a coarsest partition refinement, such that every set in the resulting partition has the same forwarding behavior across all routers. Algorithm 4 sketches how the approach from [17] creates one predicate per output port that summarizes the set of headers that are forwarded to the given port. Recall that we assume that a router is an ordered list of rules of the form $\langle tbv, p\rangle$, where tbv is a matching condition and p is the name of an output port. The result of Algorithm 4 is a disjoint partition of sets over the header space of a router.

Algorithm 4: Extracting predicates for a router

1. $Ps \leftarrow [p \mapsto \emptyset \mid p \in Ports]$
2. $seen \leftarrow \emptyset$
3. **foreach** $\langle tbv, p\rangle \in Router$ *in order of appearance* **do**
4. $\quad Ps[p] \leftarrow Ps[p] \cup (tbv \setminus seen)$
5. $\quad seen \leftarrow tbv \cup seen$
6. **end foreach**
7. $Ps[sink] \leftarrow Ps[sink] \cup \overline{seen}$

The partitions created for each router are then combined into a maximally coarse partition as follows: Let $Ps_1, \ldots, Ps_n$ be the partitions extracted from routers $1, \ldots, n$. Then the final partition can be computed using Algorithm 5.

Algorithm 5: Partition refinement

```
1 𝓡 ← {⊤}
2 foreach i = 1,...,n; p ∈ Ports do
3 |  𝓡 ← {P ∩ R, P̄ ∩ R | P = Ps_i[p], R ∈ 𝓡} \ {∅}
4 end foreach
5 return 𝓡
```

This algorithm requires an asymptotically quadratic number of BDD operations during partition refinement, a cost that is avoided with ddNFs.

First, note that each union or intersection operation (line 4 in Algorithm 4, line 5 in Algorithm 5) creates a result of size that is potentially the sum of the size of the arguments. When iterated a linear number of times, this may produce potentially quadratic space overhead. Furthermore, the number of operations in Algorithm 5 is also quadratic in the size of the result. The ddNF data-structure may likewise increase in size during an insertion. However, the overall space overhead of the ddNF structure is bounded by the number of disjoint partitions, and the number of operations for an insertion is bounded by the bit-width of the header space multiplied by the number of resulting classes (the longest path of a ddNF is at most the bit-width of the header space). There is an important constant factor that differentiates BDDs and ddNFs as well: The ternary bit-vectors in the ddNF tree can be represented using machine words. A ternary bit can be represented using two bits in the usual way: 01 for true, 10 for false, 11 for $\star$ and 00 for undefined. The intersection of two TBVs is defined if it does not contain a sequence of 00s. Then if a machine has word size w (which is typically 64 these days) one can represent a k-bit ternary bit using $\lceil 2 \cdot k/w \rceil$ words. BDDs, in contrast allocate a separate node per bit, each node has a field for the current variable and pointers to left and right children. Typical implementations use also fields for reference counts. As we show in the next section, the evaluation also shows that the ddNFs behave very well on our benchmark sets.

Atomic predicates always correspond to a union of nodes in the ddNF built from routers. This is because each atomic predicate is an intersection of DOCs corresponding to rules and each such intersection corresponds to a union of nodes in a ddNF. Thus, the number of ddNF nodes is always at least the number of atomic predicates. Our experiments show that in practice this number is pretty small, even though the worst case is prohibitive. The ddNFs originating from rules from example 1 grow exponentially. The ddNF for that example contained 8 nodes, while there are only three atomic predicates: $1{\star}{\star}, 01{\star}, 001$, but the ddNF grows exponentially with the bit-width. More generally, rules for a single router create only one atomic predicate per output port, while the number of nodes in a ddNF is potentially exponential for a fixed router. The number of atomic predicates collected for a set of routers can of course be exponential for the same reasons that ddNFs can be of exponential size.

4.4 ddNFs and DOCs and multi-dimensional prefix tries

The HSA [7] tool uses linear search over DOCs to process symbolic headers represented as TBVs and figure out the forwarding behavior for a set of packets. It does not use any specific indexing techniques to speed up matching. The Veriflow tool [8] integrates some indexing. It uses multi-dimensional prefix tries to represent rules. It is inspired by traditional packet classification data structures. Each dimension corresponds to a header field, and each trie branches on one bit at a time. The approach suggested with ddNF here would correspond to a single dimension of such a trie, or a collapsed multi-dimensional trie. One can of course create multi-dimensional structures from ddNFs, but we have not found a use for it yet. On the other hand, Veriflow uses the tries to compute the set of ports based on symbolic headers represented as TBVs. It does not pre-compute labels.

4.5 Handling rules that update packets

Let us briefly describe one way to extend using ddNFs for analysis of networks where rules can update packet headers. We limit the discussion to header transformations that have match-action rules of the form $\langle p_{in}, tbv, upd, p_{out}\rangle$, where tbv is the matching condition and upd is a ternary bit-vector, and p_{in}, p_{out} are input and output ports. A packet header t matches the rule if $t \cap tbv = t$, and it is transformed to a header $t \downarrow upd$, such that $(t \downarrow upd)[i] = (upd[i] = \star)?t[i] : upd[i]$, for each bit-position i in t. The relevant question is how to efficiently compute updates for sets of headers given by a difference of cube. If we attempt to apply rewrites on difference of cubes we quickly realize that the operations require in general to eliminate existential variables: since symbolic execution of a set of states (regardless of their representation) corresponds to working with strongest post-conditions. On the other hand, pre-conditions of guarded assignments correspond to basic substitutions with the assignment and intersections with the guard. It is therefore more convenient to close ddNFs under pre-images. The procedure for closing ddNFs under pre-images over a set of configurations $\langle doc_1, p_1\rangle, \ldots, \langle doc_n, p_n\rangle$ of DOCs and ports $p_1, \ldots, p_n$ is obtained by computing the fixedpoint under

$$wpc(\langle p_{in}, tbv, upd, p_{out}\rangle, \langle doc_{out}, p_{out}\rangle) := \langle doc_{in} \cup tbv \cap (doc_{out} \downarrow upd), p_{in}\rangle$$

Note that the operations on the resulting DOCs can be performed directly over ddNFs.

5 Experiments

We measured the efficiency of the ddNF data structure in comparison with [17] using benchmarks from the Stanford Campus Network [7] and IP forwarding tables provided from selected Azure data-centers. We used a modest laptop running Intel Core i5-3317U 1.70 GHz, 8GB Ram, running 64 bit Windows 8.1. Our implementation of [17] uses the BuDDy BDD library [15] and otherwise follows almost verbatim the presentation of [17] with one minor change. The algorithm

suggested in [17] for inserting a set P into an existing set of partitions $R_1, \ldots, R_n$ is to compute $\{P \cap R_i, \overline{P} \cap R_i \mid i = 1, \ldots, n\}$ and remove empty sets from the result. Our approach is, for each i, to first compute $P \cap R_i$. If the result is empty then we produce R_i, otherwise compute also $\overline{P} \cap R_i$ and set P to $P \cap \overline{R}_i$. We found this approach to be crucial to make the BDD based approach work.

Table 1 summarizes the comparison. We note that the ddNF-based implementation runs at least one order of magnitude faster, with the runtime being a fraction of a second even for the largest benchmark. This makes the ddNF-based approach well-suited for use with real-time updates of router rules. We also noticed that (perhaps unsurprisingly) the BDD based approach is sensitive to the initial variable order. For instance, for the Stanford benchmark set, the BDD approach is 10x faster if the initial variable order is reversed from least to most significant digit (this remains slower than ddNF, nonetheless).

	# Rules	BDD		ddNF	
		sec.	#labels	sec.	#labels
Stanford	8137	3.4	178	0.19	5149/17
DC 1	9060	2.0	829	0.05	1005
DC 2	7446	2.3	979	0.04	1157
DC 3	89871	17.7	2627	0.49	3058
DC 4	113131	29.8	3272	0.66	4077

Table 1: Measurements from five different network topologies. The Stanford benchmark is obtained from [7] and is used as a standard benchmark. We extracted only the forwarding rules from these benchmarks for our measurements. The networks DC 1-4 represent different snapshots from Azure data-centers of different size from around the globe. The numbers in the BDD and ddNF columns are time in seconds and number of generated labels. The ddNF for the Stanford network contains 5149 labels before compression and only 17 labels after line 11 of Algorithm 3. For the other networks, compression has no effect.

We also applied the ddNF data-structure in [14] as an integral part of a set of network surgeries aimed to speed up reachability queries on networks. Let us here recall main elements of our experiments there. Our experimental setup there used a Microsoft production data center located in Singapore, similar to DC 4. In more detail, the it is a fairly large switching network, with 52 core routers, each with about 800 forwarding rules (but no ACLs), and with 90 ToRs with about 800 rules and 100 ACLs each. In total, this network has about 820K forwarding and ACL rules and is a reasonable example of a complex data center.

After reducing the network with respect to the header equivalences we split forwaring rules so that each rule operates on a single header equivalence class. Then, for each such class h we compute a forwarding equivalence relation as a congruence closure relation: it is computed bottom up from reachable nodes: two nodes are equivalent with respect to a header equivalence class h if they forward h the same way. In particular, two nodes that have no forwarding rules for h are equivalent. Then, inductively, two nodes become equivalent with re-

spect to h, if the successors are pairwise equivalent. We could have opted for a stronger equivalence relation that considers two nodes forwarding equivalent with respect to h using a (co-inductive) bisimulation relation, but in the case of packet forwarding, we may expect most forwarding paths to be acyclic. Luckily, in the acyclic case, there is no difference between inductive congruence closure and co-inductive bi-simulation relations. In this way, we transformed a network with nearly a million rules to a new network with just over 10,000 rules and obtained a corresponding two-orders of magnitude speedup over analyzing the original network.

6 Conclusions

This paper developed ddNFs to quickly and incrementally decompose the header space into a much smaller set of equivalence classes. We found ddNFs an order of magnitude faster than previous approaches [17] on our benchmarks, making ddNFs especially suitable for incremental verification when router rules change rapidly.

References

1. E. Al-Shaer and S. Al-Haj. FlowChecker: configuration analysis and verification of federated OpenFlow infrastructures. In *SafeConfig*, 2010.
2. P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.
3. P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *SIGCOMM*, 2013.
4. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, Aug. 1986.
5. A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel. The cost of a cloud: Research problems in data center networks. *SIGCOMM Comput. Commun. Rev.*, 39(1), Dec. 2008.
6. P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *NSDI*, 2013.
7. P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *NSDI*, pages 113–126, 2012.
8. A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *NSDI*, 2013.
9. N. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese. Checking beliefs in dynamic networks. In *NSDI*, 2015.
10. H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with Anteater. In *SIGCOMM*, 2011.
11. N. McKeown. Mind the gap. In *SIGCOMM*, 2012. `http://youtu.be/Ho239zpKMwQ`.
12. T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. The Margrave tool for firewall analysis. In *LISA*, 2010.
13. A. Pathak, M. Zhang, Y. C. Hu, R. Mahajan, and D. Maltz. Latency Inflation in MPLS-based traffic engineering. In *Internet Measurement Conference (IMC)*, 2011.

14. G. D. Plotkin, N. Bjørner, N. P. Lopes, A. Rybalchenko, and G. Varghese. Scaling network verification using symmetry and surgery. In R. Bodík and R. Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 69–83. ACM, 2016.
15. The BuDDy BDD package. Available at `http://buddy.sourceforge.net/manual/main.html`.
16. G. G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. G. Greenberg, G. Hjálmtýsson, and J. Rexford. On static reachability analysis of IP networks. In *INFOCOM*, 2005.
17. H. Yang and S. S. Lam. Real-time verification of network properties using atomic predicates. In *ICNP*, pages 1–11, 2013.