Identifying Equivalent SDN Forwarding Behaviour

Richard Sanger
University of Waikato
rsanger@wand.net.nz

Matthew Luckie
University of Waikato
mjl@wand.net.nz

Richard Nelson
University of Waikato
richardn@waikato.ac.nz

ABSTRACT

Software-Defined Networking (SDN) enables network operators the flexibility to program their own forwarding rules, providing more than one way to achieve the same behaviour. Verifying equivalence between rulesets is a fundamental analysis and verification building block for SDN as it can be used to: (1) confirm a ruleset optimised for power efficiency or table occupancy remains equivalent, (2) verify a ruleset modified for new hardware, (3) regression test an SDN application to detect bugs early.

We present a practical and novel canonical Multi-Terminal Binary Decision Diagram (MTBDD) representation of OpenFlow 1.3 ruleset forwarding behaviour which can be trivially compared for equivalence. Basing our representation on an MTBDD provides a proven canonical form which is also compact. In this paper, we present the algorithms required to correctly flatten multi-table pipelines into an equivalent single-table, resolve equivalences in OpenFlow actions, and build the final MTBDD representation from a priority ordered ruleset. OpenFlow rulesets can typically be converted to an MTBDD within tens of seconds. We release our open-source implementation to the SDN community.

CCS CONCEPTS

• **Networks** → **Programmable networks**; • **Computing methodologies** → *Model verification and validation.*

1 INTRODUCTION

Software-Defined Networking (SDN) offers fine-grained network control by providing the ability to program low-level packet forwarding rules directly into network hardware. This fine-grained control provides an opportunity to optimise forwarding rule placement, such as optimising for power saving [14] and transforming rulesets to target different hardware pipelines [21, 22].

The flexibility enabled by SDN increases the difficulty of identifying equivalent forwarding behaviour. Equivalences occur both in the combination of matches mapping to an action, and within actions. Matches can be split into many combinations with different masks or priority orders while remaining equivalent. Actions can be reordered, applied indirectly via a group, and contain redundant operations such as a push followed by a pop while remaining equivalent. Further complicating matters, in a multi-table pipeline the forwarding behaviour observed by a single packet is represented across multiple rules which split actions and matches across multiple tables.

This paper presents a practical way to check forwarding equivalence between rulesets of two different OpenFlow 1.3 switches, enabling network operators and researchers to verify a ruleset transformation is correct. Our technique is general and can be extended to support new header fields and actions, and, with some limitations, could be adapted to other match-action packet pipelines such as P4 [7] and other OpenFlow versions. Our solution correctly handles priority ordered matches, multi-table pipelines, and complex modifications such as pushing and popping tags (e.g. VLANs) between tables. Additionally, our solution detects functionally equivalent actions, caused by redundant operations and indirection resulting from OpenFlow group actions. Our final Multi-Terminal Binary Decision Diagram (MTBDD) representation is suitable for extracting additional information about why a ruleset is not equivalent, such as identifying traffic which has different forwarding behaviour.

Section 2 defines the terminology we use, presents the OpenFlow forwarding model, shows where equivalences can occur, and introduces the options available to represent OpenFlow matches, or more generally packet-space. In Section 3 we present our method of converting rulesets into a canonical form, which has three key steps. The first step converts multi-table pipelines to an equivalent single-table, using a Cartesian product combination of tables (§3.1). The second step converts OpenFlow 1.3 actions into a canonical form representing forwarding behaviour (§3.2). The final step constructs an MTBDD from the single-table priority ordered ruleset (§3.3). This MTBDD is canonical and trivially

SOSR '19, April 3–4, 2019, San Jose, CA, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6710-3/19/04…$15.00
https://doi.org/10.1145/3314148.3314347

comparable to another. Section 3.4 identifies and resolves a rare edge case caused by redundant set-field actions.

In Section 4, we describe our implementation, and evaluate its performance. We evaluate the time to convert three rulesets into a canonical MTBDD representation. We show that our Divide-and-Conquer approach to building the MTBDD is much faster than a naive approach, reducing a 15-hour runtime to 7 minutes in an extreme case. In §5, we discuss related work and in §6, we discuss future research directions. We release our implementation to the research community [1].

2 BACKGROUND AND TERMINOLOGY

In this paper, we present our work in the context of OpenFlow 1.3 [19], the de-facto SDN standard. However, our technique applies to multi-table match-action pipelines in general.

2.1 Terminology

OpenFlow 1.3 exposes a programmable multi-table match-action pipeline. The *ruleset* installed in these tables defines the forwarding behaviour of an OpenFlow switch. We use *packet-space* to refer to a set of packets defined by the values of matchable header fields. We do not include non-matchable packet contents in packet-space as these do not influence the forwarding decision. An empty packet-space contains no packets, and a full packet-space contains all possible packets. An OpenFlow rule matches a packet-space to a set of instructions. An OpenFlow table defines instructions for the full packet-space as any unmatched packets have the default instructions applied. These instructions specify pipeline processing including actions which apply forwarding. We define *forwarding behaviour* for a packet to be the ports (if any) it egresses and all modifications made to the packet, which can vary per port. A ruleset is equivalent if the forwarding behaviour is equivalent for the full packet-space, i.e. every possible value of packet header.

2.2 OpenFlow Pipeline Processing

An OpenFlow 1.3 pipeline carries a packet with additional information: its *ingress port*, *metadata*, and *action set*. The ingress port and metadata are matchable fields which only exist within the pipeline. A developer can use metadata to store state information between tables. The action set begins empty, and a rule can clear or add actions to it throughout the pipeline. When a rule adds to an action set, it replaces any existing actions of the same type. At the end of processing, the pipeline executes the actions in the action set.

In OpenFlow 1.3 a packet begins processing in the first table. Within a table, multiple rules may match a packet. For each table, the switch finds the highest priority rule which matches the packet and applies its instructions. A switch executes instructions in the following order:

Table 1

	Priority	Match	Write Action	Apply Action
A	10	**01	Output:1	GotoTable:2
B	9	*010	Output:2	GotoTable:2
C	0	****		

Table 2

	Priority	Match	Write Action	Apply Action
D	100	1***	Clear	
E	0	****		

Figure 1: A multi-table pipeline which makes forwarding decisions in Table 1, and drops unwanted packets in Table 2.

Table 1

	Priority	Match	Write Action	Apply Action
A	100	1***		
B	0	****		GotoTable:2

Table 2

	Priority	Match	Write Action	Apply Action
C	10	*010		Output:2
D	10	*001		Output:1
E	10	*101		Output:1
F	0	****		

Figure 2: A multi-table pipeline which drops unwanted packets in Table 1, and makes forwarding decisions in Table 2.

(1) *Apply actions* first. Rules in the next table can match these modifications.
(2) If included, *clear actions* empties the action set.
(3) *Write actions* adds to the packet's action set.
(4) *Write metadata* to update the packet's metadata.
(5) If included, *goto table* sends the packet to the specified table to continue processing. Otherwise processing ends.

Finally, the action set is executed. Therefore, the forwarding behaviour of a packet through a switch is the combination of apply actions from matched rules, followed by the action set built along the way. A switch drops a packet if neither apply actions or the action set include an output action.

2.3 Difficulties in Equivalence Checking

Figure 1 and 2 both represent a simplified two table pipeline which forwards based on the lower 3 bits of the match and drops packets if their highest bit is 1. Both pipelines have equivalent forwarding behaviour. In Figure 1, rules A and B apply forwarding by adding an output action to the packet's *action set*. Rule D, in the second table, drops packets with a 1

	Priority	Match
A	100	d8:2c:07:cc:53:ed
B	100	c2:09:4c:bc:7c:e0
C	100	6b:aa:94:41:4b:a5
D	100	42:48:5e:3e:e5:16
E	100	82:e2:e6:6f:8c:b8
F	0	**:**:**:**:**:**

(a) The matches in a simple Ethernet forwarding table. Rules A-E match specific hosts to select forwarding. While the default F catches unknown hosts.

Calculation (Cumulative)	Header Space (Wildcards)		BDD (Nodes)	
	Actual	Worst Case	Actual	Worst Case
F	1	1	1	1
- A	48	48	50	50
- B	1,932	2,304	93	2,500
- C	64,649	110,592	138	125,000
- D	1,298,260	5,308,416	181	6,250,000
- E	Mem Error	254,803,968	224	312,500,000

(b) A demonstration computing the remaining packet-space reaching rule F, by subtracting all higher priority rules (A-E). We compare both actual and theoretical worst-case space complexity required for the calculation between both Header Space and Binary Decision Diagrams.

Figure 3: Space complexity when computing the packet-space reaching a default rule (F). Header Space quickly exhausts system memory after subtracting just five Ethernet addresses from a default rule. While removing duplicate wildcards greatly reduces the actual size vs. the theoretical, the scaling is still primarily dominated by exponential growth. The theoretical worst-case space complexity for a BDD is exponential. However, the actual scaling we observe is linear in memory consumption. Header Space is measured in wildcards which are a few hundreds of bytes in size and BDDs in nodes which are a few tens of bytes.

high bit match by clearing the *action set*, thus removing any output actions. The remaining packets will match either C or E, which terminate processing and apply the *action set*. The ruleset in Figure 2 drops packets in the first table. Rule A drops all packets with a 1 high bit match, and B sends the remaining packets to table 2 where rules C, D and E apply forwarding.

We emphasise that even though both rulesets have equivalent forwarding behaviour, the matches used between these rulesets are different. For the ruleset in Figure 1, rule A forwards to port 1, whereas forwarding is split into rules D and E in the ruleset in Figure 2. The rulesets also use different actions. The ruleset in Figure 1 uses *write actions* to set and then later clear forwarding, while the ruleset in Figure 2 forwards using *apply actions*.

2.4 Packet-space Representations

Representing a packet-space, i.e. an arbitrary set of packets, is a fundamental component in checking a ruleset's forwarding equivalence. However, packet space is much too large to work with uncompressed. OpenFlow 1.3 can match more than 2^{1000} unique packet headers, so we require an alternative compressed representation. Here we present representations along with their advantages and disadvantages.

OpenFlow matches are designed to map directly into Ternary Content-Addressable Memory (TCAM) so a switch can perform lookups in constant time. We refer to OpenFlow matches as TCAM-style. A TCAM-style match is a series of ternary bits (t-bits), which match either 0, 1 or * (do not care), and requires all bits to match. Any field not specified in an OpenFlow match is a do not care. TCAM-style matches are not canonical as they can be split or placed in a reordered priority while remaining equivalent. The minimisation of TCAM-style matches has been proven to be an NP-hard problem [3]. So minimisation into a canonical form is infeasible.

Header Space [15] is an alternative TCAM-style representation of matches which uses a carefully crafted bitwise encoding allowing set operations to be computed quickly using standard bitwise operations. In particular, the bitwise AND operation of two wildcards (an encoded TCAM-style match) calculates the intersection of the packet-space. Header Space combines wildcards to store more complex packet-spaces as the union of a list of wildcards, named Header Space objects. Header Space defines the set operations: union, intersection, difference, and complementation. Header Space is not canonical, though equivalence can be compared by checking if the symmetric difference between two sets is empty. Unfortunately, the difference operation between Header Space objects has a theoretical exponential worst-case expansion in the number of wildcards returned. Figure 3 shows this expansion in an example which considers the packet-space reaching a default rule F by subtracting all higher priority Ethernet address matches A-E from F. After subtracting five rules the machine ran out of memory, as the exponential growth dominates any deduplication of wildcards. While Kazemian *et al.* found lazy evaluation of difference can improve performance by allowing terms to cancel early [15], we have found it merely delays the inevitable expansion, particularly for nonequivalent rulesets where nothing cancels.

An entirely different representation of packet-space is the Binary Decision Diagram (BDD). A BDD is a directed acyclic graph that represents boolean logic [2, 17]. A BDD's layout naturally supports set operations efficiently. A BDD has a single root node. Each node has a label corresponding to the boolean variable it represents and two child branches, named low and high, corresponding with the decision made if that variable is false or true. Terminal nodes at the end of the graph represent the final decision made, either true or false. The truth of a boolean expression for a given set of input values is found by following a path through the BDD from the root node to a terminal node which holds the overall truth value. At each node along the path select the edge that corresponds to the variable's value.

A BDD most commonly refers to a Reduced Ordered BDD (ROBDD) [9] which adds restrictions to create a more condensed graph. An ROBDD is a canonical representation for a selected node ordering [9]. In practice, BDD implementations allocate all BDD subgraphs from a shared pool for efficiency [8]. This implementation detail is particularly useful when checking BDD equivalence, as equivalent BDDs are the same graph and therefore share the same root node.

In the worst case, the space complexity of any BDD operation combining two BDDs is the product of the nodes in each BDD [9]. However, in practice, Knuth [16] notes that the complexity is often closer to the sum of the nodes. Figure 3 shows our experimental results determining the packet-space which will reach the default rule F using a BDD. The theoretical worst-case scaling is exponential; however like Knuth [16], we found the actual scaling to be linear. BDD's scaling is much better suited to our usage than TCAM-style representations like Header Space, which is dominated by the worst case exponential scaling.

We used the Multi-Terminal Binary Decision Diagram (MTBDD) [10, 11] structure for the final canonical representation of a ruleset. An MTBDD extends a BDD by allowing a finite set of terminal nodes instead of only true and false. We selected an MTBDD as it solves many problems: (1) an entire ruleset can be represented as a mapping from packet-space to forwarding behaviour, (2) it is canonical and trivial to compare, (3) it supports all required operations including set operations, and (4) it performs well with a linear space-complexity in practice.

3 EQUIVALENCE CHECKING

In this section, we present our method to compare forwarding equivalence of two OpenFlow rulesets. Our process has three key steps: (1) flattening multi-table pipelines to an equivalent single-table representation using Cartesian product merging of rules (§3.1), (2) converting the actions applied by the flattened rules to a canonical representation of forwarding behaviour (§3.2), (3) building a canonical representation of packet-space mapped to the canonical forwarding behaviour using an MTBDD (§3.3).

Algorithm 3.1 Flatten OpenFlow Tables to a Single-Table Equivalence

```
Input: $Tables$ Lists of original rules per table
Output: $T_s$ The resulting single table equivalence
1:  function FLATTEN_TABLES($first$, $TableIndex$)
2:      $ST \leftarrow$ Empty Table/List
3:      for all $second \in Tables[TableIndex]$ do
4:          $merged \leftarrow$ MERGE($first$, $second$)    ▷ MERGE as per description in § 3.1
5:          if $merged$ != NULL then
6:              if $merged.GotoTable$ then
7:                  $ST$.add(FLATTEN_TABLES($merged$, $merged.GotoTable$))
8:              else
9:                  $ST$.add($merged$)
10:             end if
11:         end if
12:     end for
13:     return $ST$
14: end function
15: $first \leftarrow$ An Empty Rule
16: $T_s \leftarrow$ FLATTEN_TABLES($first$, 0)
17: return $T_s$
```

This final MTBDD representation makes it trivial to check the forwarding equivalence of two rulesets. Additionally, we show how it is possible to perform other operations on this representation such as finding the packet-space with different forwarding behaviour. However, one rare edge case remains where equivalences may not be detected due to redundant set-field actions, which we resolve in §3.4.

3.1 Convert to a Single-Table Equivalence

By definition, we need to flatten a ruleset's multiple tables in order to form a canonical representation. Therefore, we first flatten a multi-table ruleset to an equivalent single-table ruleset using the recursive approach described Algorithm 3.1. FLATTEN_TABLES recursively computes the Cartesian product of all rules, simulating all paths a packet can take through the pipeline. Rules in a table are MERGED with all the rules in the next table they *goto* recursively until only one table remains. The MERGE operation creates a single rule which matches only packets reaching both rules and applies the equivalent combined forwarding behaviour. The MERGE operation is associative. For efficiency, we combine rules in table order to avoid computing unreachable paths. Similar techniques have been used in prior work [21, 22] as part of transforming a ruleset to fit a fixed-function pipeline, but neither can reconcile stacked tags (e.g. VLAN, MPLS, and PBB). Our

Algorithm 3.2 Merging OpenFlow Matches (bitwise)

```
Input: $W_f$ A t-bit written by the first rule
Input: $M_f$ A t-bit of the matches of the first rule
Input: $M_s$ A t-bit of the matches of the second rule
Output: $M_n$ The new merged match as a t-bit
1: function MERGE_BITWISE($W_f$, $M_f$, $M_s$)
2:     if $W_f = *$ then
3:         $ps \leftarrow M_f$
4:     else
5:         $ps \leftarrow W_f$
6:     end if
7:     if $ps \cap M_s = \varnothing$ then
8:         return NULL        ▷ No overlapping packet-space
9:     end if
10:    if $W_f = *$ then
11:        $M_n \leftarrow M_f \cap M_s$
12:    else
13:        $M_n \leftarrow M_f$
14:    end if
15:    return $M_n$
16: end function
```

approach is more general as it works with packet-spaces and handles stacked tags.

Next, we outline how to MERGE the individual components of a rule: match, write actions, apply actions, and priority, with another rule, such that the Cartesian product MERGE of all tables results in an equivalent single table representation. We say the first rule is MERGED with the second rule in the next table. The MERGED match is the most complicated to calculate as it depends on the *apply action* modifications made by the first rule.

Merging matches: As the merged rule will replace the first rule, the merged match must represent the packet-space as it enters the first rule that is also accepted by the second rule. Naively, this is the intersection of the matches, but this does not account for any modifications the first rule makes to the packet. Modifications will not be present in the ingress packet and therefore should not be included in the merged match, but we must check the modified packet is accepted by second rule's match; otherwise, the packets cannot reach both rules, so we do not generate a rule. We consider two types of modifications: (1) the first rule sets a header field matched by the second rule, and (2) the first rule either pushes or pops a tag on the packet.

For example, consider merging the first rule Match={Src IP:1.1.1.1} and Apply Actions=[Set Src IP:2.2.2.2, POP_VLAN] with the second rule Match={VLAN_VID:100, Src IP:2.2.2.2}. The merged match is {Src IP:1.1.1.1, VLAN_VID1:100}. Notice that even though the first rule rewrites the Src IP, the merged match inherits its match from the first rule. However, the rewritten IP is simulated to check that it matches the second rule. Additionally, the merged match matches the inner VLAN as the second rule matches the VLAN after a pop operation and that field's value had not been modified.

Merging matches with overwritten fields: Consider merging two rules where the first rule modifies a field that the second rule matches using *apply actions* or *write metadata*. Algorithm 3.2 shows the bitwise operation to calculate a merged match for one t-bit. Where M_f and M_s are one t-bit of the first and second matches and W_f is the t-bit written by the first *apply actions* (* if not modified). Lines 2-6 simulate the value of the t-bit reaching the second rule, and 7-9 check that the intersection between the packet and second match is not empty, i.e. packets can hit both rules. Lines 10-14 determine the ingress packet-space that will be accepted by both rules. If the first rule wrote to the t-bit, this change is not present in the ingress packet, so the merged match is the match from the first rule. Otherwise, the ingress packet is unchanged between rules, so the merged match is the intersection of both matches. In practice, OpenFlow 1.3 always sets the value of an entire field so the bitwise operation can be applied per header field, expect for *metadata* where rules can set arbitrary bits.

Merging matches with multiple tags: Consider merging rules where the first pops or pushes a tag. These actions offset the tag reaching the second rule. In OpenFlow 1.3 there are three tags which can be pushed and popped: VLAN, MPLS, and PBB [19]. A single rule can only match the outermost tag, and matching inner tags is only possible in a multi-table pipeline by first popping the outer tag and matching the inner tag, which is now the outer, in the next table. In order to represent matching an inner tag in our single-table equivalence, we create new match fields to express matching the n^{th} tag, allowing all multi-table pipelines to be expressed in a single table. For example, we replicate the VLAN fields VLAN_VID and VLAN_PCP which match the outermost VLAN tag to create VLAN_VID1 and VLAN_PCP1 to match the next tag.

Algorithm 3.3, CALCULATE_EGRESS, shows how to calculate the packet-space of packets leaving a rule destined to another table for further processing. CALCULATE_EGRESS combines both the packet-space restrictions imposed by the match and the actions applied to the packet and calculates the egress packet-space. CALCULATE_EGRESS is used by Algorithm 3.4 to calculate the final merged match.

We define two helper methods to renumber tag fields PROMOTE_FIELD($match$, $field$, rep) and DEMOTE_FIELD(...). If $match$ is updated in-place, $field$ selects the field to promote or demote and rep the number of times to repeat. PROMOTE_FIELD pops the outermost tag's fields by decreasing the number of the field, e.g. FIELD is removed, and *FIELD1*

Algorithm 3.3 A rule's egress packet-space (tag aware)

```
Input: rule An OpenFlow rule object
Input: tag An object containing information about a tag
    including the tag.push and tag.pop action and tag.fields
    which are part of the tag
Output: egress_packet The packet-space leaving the rule
Output: written_fields The packet-space overwritten by
    set_field actions.
Output: offset The offset the tag vs. the ingress packet.
 1: function CALCULATE_EGRESS(rule)
 2:     rewritten ← full packet-space
 3:     egress_packet ← copy(rule.match)
 4:     op_count ← 0
 5:     for all act ∈ rule.instructions.apply_actions do
 6:         if act.type = set_field then
 7:             egress_packet[act.field] ← act.value
 8:             rewritten[act.field] ← act.value
 9:         else if act.type = tag.push then
10:             for all field ∈ tag.fields do
11:                 DEMOTE_FIELD(egress_packet, field, 1)
12:                 DEMOTE_FIELD(rewritten, field, 1)
13:                 offset ← offset - 1
14:             end for
15:         else if act.type = tag.pop then
16:             for all field ∈ tag.fields do
17:                 PROMOTE_FIELD(egress_packet, field, 1)
18:                 PROMOTE_FIELD(rewritten, field, 1)
19:                 offset ← offset + 1
20:             end for
21:         end if
22:     end for
23:     return egress_packet, rewritten, offset
24: end function
```

becomes *FIELD*. DEMOTE_FIELD pushes an outer tag field by increasing the number of the field, e.g. *FIELD* becomes *FIELD1*. The value of the newly added *FIELD* is set to the value of *FIELD1* because OpenFlow specifies the new value for a pushed field is taken from the outermost header.

CALCULATE_EGRESS is simplified to process only a single generic *tag*; our implementation repeats this process for all tags, i.e. VLAN, MPLS, PBB. CALCULATE_EGRESS line 23 returns the packet-space leaving the rule destined to another table, the portion of that packet-space that has been written by *apply-actions*, and the final offset of the tag compared to the packet entering the rule. Both the egress packet-space and the rewritten packet-space are calculated near identically with the exception that the egress packet-space is initially copied from the first rule's matches, whereas the written fields begin as the full packet-space. The main for loop, lines 5-22, simulates all modifications made by *apply-actions*.

Algorithm 3.4 Calculating the merged match of two rules

```
Input: first The first rule
Input: second The second rule
Output: merged_match A new match
 1: function MERGE_MATCH(first, second)
 2:     egress_packet, rewritten, offset ← CALCULATE_EGRESS(first)
 3:     for each match bit do:
 4:         merged_match ← MERGE_BITWISE(rewritten, egress_packet, second.match)
 5:     end for
 6:     if offset >= 1 then
 7:         for all field ∈ tag.fields do
 8:             DEMOTE_FIELD(merged_match, field, offset)
 9:         end for
10:     else if offset <= −1 then
11:         for all field ∈ tag.fields do
12:             PROMOTE_FIELD(merged_match, field,-offset)
13:         end for
14:     end if
15:     return first.match ∩ merged_match
16: end function
```

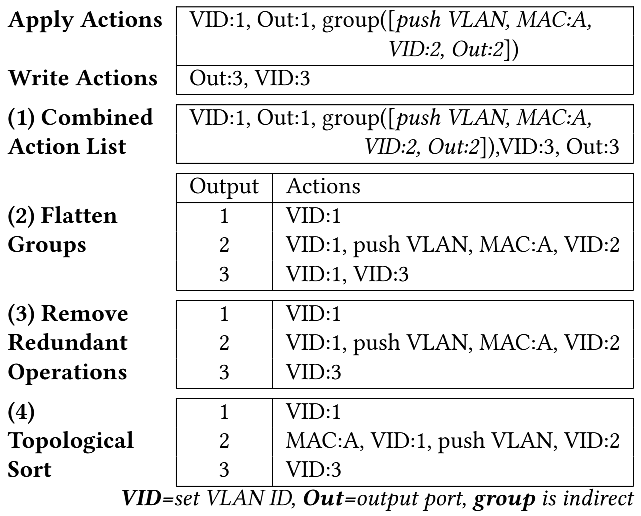
On encountering any set-field action the field in the egress packet-space is assigned the new value, even if not tag related. Whereas on encountering a push or pop operation either DEMOTE_FIELD or PROMOTE_FIELD is called to offset all tags. Because a tag might include multiple matchable fields, lines 10-14 and 16-20 apply PROMOTE or DEMOTE to all fields within the tag. The packet-spaces returned represent the packet the next rule in the pipeline sees with tags offset appropriately.

Algorithm 3.4, MERGE_MATCH, takes two rules and returns the match for the final merged rule. First, line 2 takes the result from CALCULATE_EGRESS (Alg 3.3) of the first rule, then lines 3-5 apply the bitwise merge (Alg 3.2) between the first rule's egress packet-space and the second rule's match. This merge checks that packet-space reaches the second rule and returns the merged packet-space. However, this merged packet-space applies to the modified packet, not the ingress packet, thus we revert the tag locations back to the ingress offset (lines 6-14). Finally, because PROMOTE_FIELD removes popped tags, matches on these popped tags are lost and need to be added back into the merged match. Line 15 restores any matches on popped tags by taking the intersection of the first rule's match and the calculated *merged_match*, and returns the result.

Merging write actions: *Write actions* are merged as if they were *action sets* following OpenFlow pipeline processing; first applying *clear action* instructions and then overwriting existing actions with those written later in the pipeline.

Original	Minimal	Notes	Resolved By
group([output:1])	output:1	groups add indirection when outputting packets	Step (2)
output:1, set vlan-vid:1	output:1	changes made to a dropped packet are irrelevant	Step (2)
set vlan-vid:1, set vlan-vid:2, output:1	set vlan-vid:2, output:1	an overwritten set-field is redundant	Step (3)
set vlan-vid:1, pop vlan, output:1	pop vlan, output:1	fields set on popped headers are redundant	Step (3)
push vlan, pop vlan, output:1	output:1	push+pop pairs are redundant	Step (3)

Table 1: Base-cases of equivalent operations in OpenFlow 1.3 we identified, showing an example of the original action set, compared to a minimal representation and the step in our process which resolves them.



Apply Actions	VID:1, Out:1, group([*push VLAN, MAC:A, VID:2, Out:2*])	
Write Actions	Out:3, VID:3	
(1) Combined Action List	VID:1, Out:1, group([*push VLAN, MAC:A, VID:2, Out:2*]),VID:3, Out:3	
	Output	Actions
(2) Flatten Groups	1	VID:1
	2	VID:1, push VLAN, MAC:A, VID:2
	3	VID:1, VID:3
(3) Remove Redundant Operations	1	VID:1
	2	VID:1, push VLAN, MAC:A, VID:2
	3	VID:3
(4) Topological Sort	1	VID:1
	2	MAC:A, VID:1, push VLAN, VID:2
	3	VID:3

***VID**=set VLAN ID, **Out**=output port, **group** is indirect*

Figure 4: Canonicalisation of a complex action list.

For example, the *write actions* of the first rule [set vlan_vid:10, set vlan_pcp:5] and the *write-actions* of the second rule [set vlan_vid:20] are merged as [set vlan_vid:20, set vlan_pcp:5]. In this case, the first rule's write of vlan_vid is overwritten by the second rule, and the vlan_pcp write remains from the first *write-actions* as it is not overwritten by the second.

Merging apply actions: To ensure that all actions are applied in the order a packet traverses them, *apply actions* are concatenated in pipeline processing order.

Merging priority: When flattening tables, relative processing priority must be maintained, e.g. in Figure 1 the priority order of merged rules from highest to lowest is: A+D, A+E, B+D, B+E, C+D, and C+E. The relative priority order of the first rule takes precedence over the second rule. We achieve this by scaling all priorities based on the table that they are installed into to allow enough space between two adjacent priorities to fit all priorities of subsequent tables, using this formula $new\_priority(r) = priority(r) \times (MaxPriority^{|tables|-1-table_{index}(r)})$. These scaled priorities are merged using addition.

3.2 Identify Equivalent Actions

Now we have a single-table representation of the original ruleset, we look to form a canonical representation of the forwarding behaviour applied by each rule. Identifying equivalent actions is difficult because OpenFlow has many ways to represent the same behaviour. We read the OpenFlow 1.3 specification and identified five base cases of equivalent action sequences (listed in Table 1), which we resolved by conversion to a minimal form.

We represent actions per output port minimised and ordered to create a canonical form. We are careful to perform this canonicalisation in a dependency-aware manner to ensure actions are not removed or reordered in a way that changes forwarding behaviour. We define a dependency between two actions if performing them in reverse order will result in different forwarding behaviour. For example, every set-field is dependent on itself, setting VLAN-VID is additionally dependent on push and pop VLAN operations, and all actions share a dependency with output actions. Our process of converting actions to this canonical format is detailed below and is shown by example in Figure 4.

(1) We combine a rule's *write actions* and *apply actions* into a single apply actions list. The *write actions* are appended to the end of *apply actions* in the processing order detailed by OpenFlow [19]. Step 1 of Figure 4 shows an example output of this step.
(2) We flatten all groups and output actions by creating a mapping of the output port to actions (Step 2 of Figure 4). The process walks from the start of the combined action list collecting the actions applied to the packet. When an output action is found, instead of collecting it we map the output port to the actions collected thus far. When a group is encountered, we make a copy of the actions collected so far for each group bucket. Then, we walk each bucket removing the group while preserving its behaviour.
(3) We walk through each list finding and removing redundant actions. These include duplicate set-fields, and sequences such as push VLAN, set VID, pop VLAN. Step 3 of Figure 4 removes the redundant VID:1 action

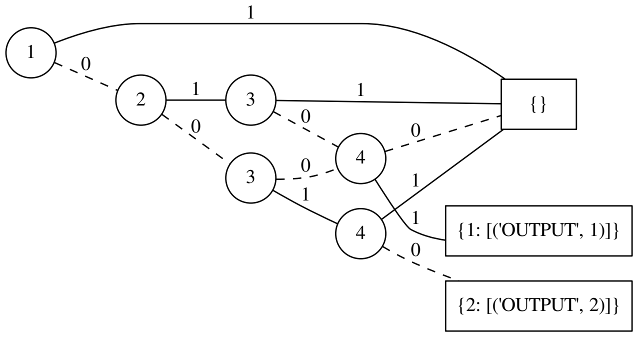


Figure 5: The canonical MTBDD representation of the equivalent rulesets shown in Figures 1 and 2. The low (dotted) edge is the decision made if a bit in the header is 0, and the high (solid) edge is the decision made if a bit is 1.

on output 3. These cannot be removed if another dependency of a different type is found in-between, as seen with output 2; the push VLAN action between the VID stops VID:1 being removed as VID:1 refers to a different VLAN header to VID:2.

(4) We perform a topological sort on the list to normalise ordering while maintaining dependency ordering. Step 4 of Figure 4 shows how **M**AC:A is sorted alphabetically before **V**ID:1. But, a dependency exists between **V**ID:1 and **P**ush VLAN which prohibits reordering.

The process normalises OpenFlow groups to flat representations of output actions, minimises the result by removing redundant actions, and finally sorts the result while maintaining dependencies. The result of this process is a minimal canonical representation of forwarding behaviour through an OpenFlow pipeline.

3.3 Resolve Equivalent Matches

At this point in the process we have a single-table of overlapping priority-ordered matches, each mapping to a canonical representation of its forwarding behaviour. Now we must check the forwarding for the complete packet-space is the same. For this, we use an MTBDD [10, 11] as it provides a suitable canonical mapping from packet-space to forwarding behaviour as we discussed previously in Section 2.4.

Figure 5 shows a complete canonical MTBDD representation of the rulesets in Figures 1 and 2. Each node is numbered corresponding to a bit in the packet header with 1 being the most significant bit used for dropping packets. The branches from each node represent the forwarding decision made if that bit is 0 or 1 for any given packet, the terminal (leaf) node holds the forwarding decision. We use a special terminal node ∅ to represent empty packet space, allowing an

Algorithm 3.5 Convert a Rule to a BDD

```
Input: nodes The node cache, indexed by [num, low, high]
Input: terminals The terminal cache, indexed by [action]
Input: bits Match as t-bits                    ▷ 0, 1 or *(don't care)
Input: action A canonical form of the rule's actions
Output: BDDRoot The resulting BDD representation
  $BDDRoot \leftarrow terminals[action]$
  $numBits \leftarrow |bits|$
  for $i = numBits - 1$ to 0 do
      if $bits[i] = 0$ then
          $BDDRoot \leftarrow nodes[i, BDDRoot, \varnothing]$
      else if $bits[i] = 1$ then
          $BDDRoot \leftarrow nodes[i, \varnothing, BDDRoot]$
      end if                    ▷ $bits[i] = *$ does not add a node
  end for
```

MTBDD to represent a partial packet-space. MTBDD equivalence is trivial to check. Equivalent MTBDDs have identical root nodes because efficient implementations maintain only one instance of every subgraph [8].

Converting an OpenFlow rule to a partial BDD: First, OpenFlow match fields are concatenated together in a consistent order and fields not included in the match are filled with * to form a TCAM-style match. The 40 matchable fields in OpenFlow 1.3.5 are represented in 1261 t-bits. Algorithm 3.5 shows the conversion to a BDD; the result is a BDD representing a partial packet-space defined for packets matching the rule with the remaining packet-space empty (∅). We build the BDD in reverse from the terminal node up to the root for efficiency. This ordering is more efficient because to add a node within a BDD requires its parent node to be recreated with the correct child reference, which has the flow on effect of recreating all nodes until the root. Building top down requires all nodes to be recreated for each insertion. Whereas adding a node to the root of a BDD requires only the creation of one new node.

Only 0 and 1 t-bits add nodes to the BDD, * does not create a node as it does not influence the decision made. The node ordering chosen within the BDD will change the BDD size; however, finding the best node ordering is NP-complete [6]. We found using a node ordering so that the most significant bit of a field is stored in the lowest numbered node (i.e. top of the graph) more efficiently stores prefix matches than the reverse ordering.

Converting a priority ordered table to a BDD: Representing individual rules as partial BDDs is not sufficient, as this ignores the priority order in OpenFlow tables. Algorithm 3.6 details the process of converting a priority ordered list of partial BDDs to a full BDD for the complete packet-space. The intuition is that lower priority rules can only match the packet-space not already represented in higher

Algorithm 3.6 Convert an OpenFlow Table To a BDD (Naive)

```
Input: nodes The node cache, indexed by [num, low, high]
Input: rules A priority ordered list of rules represented as
    BDDs Algorithm 3.5
Output: BDD A full representation of forwarding behaviour
 1: $BDD \leftarrow \emptyset$
 2: for all $r \in rules$ do
 3:     $BDD \leftarrow$ PRIORITYADD$(BDD, r)$
 4: end for
 5: function PRIORITYADD$(f, s)$
 6:     if $f = \emptyset$ then
 7:         return $s$
 8:     end if
 9:     if ISTERMINAL$(f)$ or $s = \emptyset$ then
10:         return $f$
11:     end if                    ▷ Note: $terminal.num > node.num$
12:     if $f.num = s.num$ then
13:         return $nodes[f.num,$PRIORITYADD$(f.low, s.low),$
14:                         PRIORITYADD$(f.high, s.high)]$
15:     else if $f.num < s.num$ then
16:         return $nodes[f.num,$ PRIORITYADD$(f.low, s),$
17:                         PRIORITYADD$(f.high, s)]$
18:     else                                   ▷ $f.num > s.num$
19:         return $nodes[s.num,$ PRIORITYADD$(f, s.low),$
20:                         PRIORITYADD$(f, s.high)]$
21:     end if
22: end function
```

Algorithm 3.7 Convert an OpenFlow Table To a BDD (D&C)

```
 1: while $|rules| > 1$ do
 2:     $newRules \leftarrow List()$
 3:     for all $r1 \in$ EVEN$(rules)$; $r2 \in$ ODD$(rules)$ do
 4:         $newRules.append($PRIORITYADD$(r1, r2))$
 5:     end for
 6:     if $mod(|rules|, 2) = 1$ then
 7:         $newRules.append(rules[-1])$
 8:     end if
 9:     $rules \leftarrow newRules$
10: end while
11: $BDD \leftarrow rules[0]$
```

priority rules. As such, lower priority rules can only fill empty (∅) space in the BDD. Lines 1-4 add each rule to an empty BDD from high to low priority; PRIORITYADD takes the highest priority as its first argument. The PRIORITYADD function recursively walks all nodes in both graphs in unison until leaves are found (lines 12-20), and is the common basis of all BDD operations, called the APPLY operation. Lines 6-11 define the PRIORITYADD operation we APPLY. If at any time the first (higher priority) BDD becomes empty the second BDD (lower priority) will be returned, otherwise reaching a terminal node on the first or an empty node on the second will return the first.

In our evaluation (§4.2) we find Algorithm 3.6 can be slow, because (1) applying the *priorityAdd* operation to rules from highest to lowest priority adds one small BDD to an ever-growing BDD, and (2) adding nodes to the bottom of a BDD requires all nodes above to be rebuilt. Performance can be improved by reordering the PRIORITYADD operations; provided that only priority adjacent BDDs are combined, the final BDD is the same.

For better performance we created a Divide-and-Conquer (D&C) approach shown in Algorithm 3.7, which replaces lines 2-4 of Algorithm 3.6. Algorithm 3.7 works similarly to merge sort, where even and odd numbered rules in the list are combined pairwise (lines 3-5) resulting in a list half the size repeatedly until one final BDD remains as checked by line 1. Lines 6-8 check if the list of BDDs is uneven and will add the remaining BDD, which has no pair, to the new list. In the D&C approach, most PRIORITYADD operations combine two small BDDs, limiting the number of nodes that have to be subsequently rebuilt, thus improving performance.

Finding different forwarding behaviour: In order to identify the packet-space, which can, in turn, identify the OpenFlow rules resulting in different forwarding behaviour, we define the difference operation using the APPLY operation as a base. This is logically similar to a set difference, however, in an MTBDD there are multiple options for constructing the terminal node. We define a simple yet powerful version of difference that returns a normal BDD where the difference is mapped to the true terminal and matching packet-space to false. Our implementation maps differences back to the original rules. Another useful difference operation of two MTBDDs returns both actions encoded in the terminal node for packet-space which differs, otherwise ∅. Encoding both actions in the terminal is useful for analysing the actions without needing to convert back to flow rules, and we use this difference in the **lazy** action checking described next.

3.4 Resolve redundant set-field actions

In our method described so far we have transformed an OpenFlow ruleset into a seemingly canonical MTBDD representation of forwarding behaviour. However, false negatives can arise in a rare edge case because our method of converting actions to a canonical form assumes independence to the packet header. In most cases this is a correct assumption; however, a subtle edge case exists where an action sets the value of a field back to its original value in the packet header as this is equivalent to not modifying the field.

	Match IP Dst.	Write IP Dst.
A_1	1.1.1.0/31	1.1.1.1

(a) Always set the field to 1.1.1.1

	Match IP Dst.	Write IP Dst.
B_1	1.1.1.1/32	
B_2	1.1.1.0/31	1.1.1.1

(b) First ignore packets which are 1.1.1.1

	Match IP Dst.	Write IP Dst.
C_1	1.1.1.0/32	1.1.1.1
C_2	1.1.1.0/31	

(c) First set-fields which are not 1.1.1.1

Figure 6: Three equivalent rulesets which demonstrate that canonical actions can be dependent on the match, because setting a field is redundant if the field already has the same value. Rules are ordered from highest priority to lowest.

Figure 6 shows three equivalent rulesets which highlight the flaw with this assumption. The equivalence of these rulesets is simple to reason about, as we only need to consider packets with two input IP destinations, 1.1.1.0 and 1.1.1.1, which are both output as 1.1.1.1. Figure 6a rule A_1 rewrites both 1.1.1.0 and 1.1.1.1 to 1.1.1.1. Figure 6b rule B_1 matches 1.1.1.1 without rewriting, B_2 matches 1.1.1.0 and rewrites it to 1.1.1.1. Figure 6c rule C_1 matches 1.1.1.0 and rewrites it to 1.1.1.1 and C_2 matches 1.1.1.1 without rewriting. If we compare actions, Figure 6b and 6c take no action on 1.1.1.1, whereas Figure 6a rewrites it to 1.1.1.1 and as such would incorrectly be found nonequivalent.

We next detail both an eager and lazy solution to this problem. We have implemented a prototype of both solutions and found they correctly resolve equivalences.

Eager: This approach ensures the resulting MTBDD is canonical. As we use a minimal representation to form our canonical action representation, we should exclude a set-field action when the packet already has that field set to the same value because it is redundant. In this solution, we replace rules that contain set-field actions with a sequence of rules without the set-field for packets already set to that value before building the MTBDD. The logic is to create a copy of every rule containing set-field actions with 1) a specific match on the written value set, and 2) the set-field instruction removed. This new rule is placed at a higher priority and provides a minimal representation for the actions. This process must be applied to all combinations of set-fields, making it scale with the number of unique set-fields. Figure 7 shows how to convert the rule in Figure 7a into the four priority-ordered rules in Figure 7b with a canonical action set for all combinations of set-field values.

Match	Set-Fields
—	EthDst:A, EthSrc:B

(a) Original rule

Match	Set-Fields
EthSrc:A, EthSrc:B	—
EthSrc:B	EthDst:A
EthDst:A	EthSrc:B
—	EthDst:A, EthSrc:B

(b) A priority-ordered ruleset, using minimal action sets for cases where the packet already contains the value set by a set-field action.

Figure 7: The conversion from a single rule containing multiple set-field actions into all possible combinations where those set-fields can be eliminated as the packet header already contains the value, filtered by adding it as a match. This minimises the action set for those cases allowing detection of equivalent rulesets where are actions are dependant on the matched value as shown in Figure 6.

A set-field action in OpenFlow 1.3 sets the entirety of that field to the value supplied, so our eager solution scales exponentially with the number of set-field actions, $2^{|setfields|}$. Anecdotally we have found this exponential scaling is not a problem as the number of set-field actions is generally low. However, OpenFlow 1.5 allows a programmer to set partial fields (i.e. individual bits), if we apply the same solution, then combinations need to be made per bit, which becomes infeasibly large.

Lazy: in this approach, the MTBDD is built as usual, and the comparison is modified to include additional checks lazily. The comparison first checks if the two MTBDDs are already equivalent and can break early. Otherwise, the equivalence check performs additional checks on the differing portions. This additional checking can break early at the first non-equivalent portion it finds as this means the forwarding behaviour as a whole is not equivalent. Therefore, the next portion only needs to be evaluated when redundant set-fields cause the difference.

To perform this additional check, we leverage the structure of the MTBDD by APPLYING a difference operation which upon finding a difference encodes a terminal that includes both actions. A single path through this MTBDD from root to terminal encodes a portion of the mismatched packet-space to the two action different actions. Any action which omits a redundant set-field will always become a separate path in the MTBDD. The path is different because a ruleset must include a set-field in the actions for all other values of the field apart from the redundant. Therefore the actions

	Original		To Single-Table		To MTBDD		
Ruleset	No. Rules	No. Tables	No. Rules	Time	Naive Alg. Time	D&C Alg. Time	No. Nodes
Faucet Router	582	8	11,447	9.71s	1.97s	1.13s	130,287
Faucet Access	1937	8	7,216	6.77s	0.82s	0.64s	74,148
RouteViews FIB	740,332	1	740,332	0.6s	23.82s	22.66s	279,985
FIB Reversed	740,332	1	740,332	0.6s	15.09hr	425s	10,009,281

Table 2: Details of the rulesets evaluated, and the MTBDD build time for each. The time taken to convert the ruleset to a canonical MTBDD format is divided into the time to convert to a single-table, and time to build the MTBDD using both the Naive and Divide-and-Conquer approach.

are different and will be stored in different terminal nodes. The additional check converts each path to the match it represents, and then each matched field is added as a set-field to the beginning of both rulesets' apply actions and then the actions are canonicalised as per Section 3.2 and compared again. Rerunning canonicalisation removes any duplicate set-fields we introduced. This comparison would be equally valid if it removed rather than added set-fields.

Compared to the eager approach the lazy approach will add near zero overhead unless rulesets differ due to redundant set-fields. This lazy approach can be applied per bit (rather than field) to deal with cases like OpenFlow 1.5 which supports bitwise set-fields, without additional overhead. Further, it avoids the expansion issue of the eager approach. However, unlike the eager approach, the lazy approach does not result in a canonical MTBDD, so other MTBDD operations may need to consider this case.

4 EVALUATION

4.1 Implementation

We implemented our ruleset equivalence checking method in Python [1]. We accept Ryu [20] rulesets. To represent packet-space internally we use and convert between OpenFlow, Header Space, and BDD where appropriate. Building a single-table equivalence uses the OpenFlow and Header Space [15] packet-space representations which support quick intersection operations. While most of the code is implemented in Python, some hot spots have been converted to C. The MTBDD is implemented entirely in C with Python bindings for better performance. We used the CUDD BDD library [25] as a base for our MTBDD and added our custom rule conversion, PRIORITYADD and difference operation logic.

4.2 Performance

In order to evaluate our solution, we present its performance with three different real-world rulesets. We do not evaluate the time to check equivalence of any two rulesets as equivalent rulesets have the same memory address for their root nodes [11, 25] and the additional lazy check is trivial to compute. Instead, we measure the time to build the MTBDD representation. We perform our tests with an i7-4790 @3.6Ghz and 8GB of RAM; our implementation is single-threaded.

We evaluate the rulesets shown in Table 2. These consist of two captures, Faucet Router and Faucet Access, from two OpenFlow switches in a real-world enterprise deployment [26] which were programmed by the Faucet [5] controller. The Faucet controller was configured to perform VLAN switching, IPv4/6 routing, and stateless firewalling. Faucet Router has more complexity than Faucet Access as it was connected to the upstream and carries routes. Faucet Access does not carry routes, but had a larger ruleset due to having more ports, each with a stateless firewall policy applied. RouteViews FIB is based on a RouteViews [18] RIB[1], which we converted to a FIB. FIB reversed is the same as RouteViews FIB but with the bit order reversed so that the least significant bit is the lowest numbered node in the BDD. This demonstrates performance in the case where a poor node ordering is chosen, as is evidenced by the increase in the final size of BDD from 280K to 10 million unique nodes.

Table 2 shows the time in seconds it takes to convert each OpenFlow ruleset into an MTBDD for equivalence checking and the final size of the MTBDD in unique nodes. We report the conversion to an MTBDD in two parts, first converting a multi-table pipeline to an equivalent single-table (§3.1) and second the time to build this table into a canonical MTBDD (§3.2, §3.3). We compare results for both the Naive MTBDD Algorithm 3.6 and the Divide-and-Conquer (D&C) approach Algorithm 3.7.

For the Faucet rulesets conversion to a single-table is the most expensive operation. This is not surprising as this operation is implemented primarily in Python, while the conversion to a BDD is primarily performed in C. The D&C approach outperforms the Naive approach in all cases. We believe this better performance is because while both the Naive and D&C approaches perform the same number of PRIORITYADD operations, the size of the BDDs added are

[1]RouteViews RIB available: http://archive.routeviews.org/oix-route-views/2018.02/oix-full-snapshot-2018-02-13-0000.bz2

on average smaller for D&C. Anecdotally we also expect rules with similar match fields to have the same priority and compute faster when added to a BDD compared to rules with different fields which, if added to the bottom of the BDD, require all parent nodes to rebuilt. The performance of our implementation is good with all rulesets except for the intentionally poorly ordered FIB reversed. While not quite fast enough for real-time applications, it is fast enough to be added to a controller's regression testing, or offline checking of potential ruleset optimisations. Comparing FIB reversed, the difference in build time between the Naive approach of 15 hours and D&C's 7 minutes is significant and shows that D&C will finish in a reasonable time even if a poor BDD node ordering is selected. This removes the need to pick the most optimal node ordering, which is an NP-complete problem [6].

4.3 Completeness

Our solution will not return false positives, but, in rare cases can return false negatives, i.e. equivalent rulesets may incorrectly be deemed nonequivalent as there remain equivalent actions we have not resolved as they require additional information. For example, equivalent actions can arise with the processing of special OpenFlow ports, such as a FLOOD output which is equivalent to an output action for every port on the switch. The correctness of resolving such cases depends on the switch's state, such as the number of ports connected to a switch. The validity of considering such cases equivalent is unclear, as a state change would invalidate a result, and ultimately depends on the use case.

This issue is not fundamentally unsolvable, it just requires a developer to design a minimisation as we did in Section 3.2. We have not implemented equivalences which require additional switch state, but we highlight these as depending on the use case they may be important.

5 RELATED WORK

Equivalence Checking: Yang *et al.* [28] presented two algorithms to compare OpenFlow ruleset equivalence, but did not implement them. Their first, the *match-field oriented approach*, considered a rule at a time from the first ruleset, and successively eliminated matching rules from the second ruleset. If all rules were eliminated, then the rulesets were equivalent. However, it did not account for overlaps in rules which, when encountered, only removed the highest priority match; the remaining shadowed rule was not eliminated, causing equivalent rulesets to be deemed nonequivalent. Their second approach, the *action oriented approach*, created a canonical mapping of action to matches. Yang *et al.* did not consider equivalence in actions, and as they did not implement their algorithms, did not specify a particular match representation. From our experience, a BDD representation would be suitable. Our implementation using an MTBDD provides a practical canonical match to action mapping, and shows how equivalences can exist in actions and be resolved.

BDDs in networking: Smolka *et al.* [24] and Arashloo *et al.* [4] used BDD structures to compile high-level languages into physical network topologies. Smolka *et al.* introduced the Forwarding Decision Diagram (FDD), and Arashloo *et al.* extended the FDD to encode stateful operations. Both found the FDD structure is efficient and offers chances for optimisation through removing equivalences. An FDD differs from an MTBDD as a decision node considers an entire value of the field rather than a single bit and cannot represent the arbitrary masking found in OpenFlow. Yang and Lam [27] used a BDD representation of packet-space to create atomic predicates to verify forwarding behaviour, including reachability, loop detection, and black hole detection. They found their approach was fast enough to run in real-time. BDDs and MTBDDs have also been used to represent firewall ACL rules for verification checking [12, 29] and as a structure for fast packet classification [12, 13, 23].

While the use of BDDs to represent aspects of forwarding behaviour is not new, our work highlights novel considerations required to use MTBDDs for equivalence checking of OpenFlow forwarding behaviour, including resolving equivalences in actions which prior work ignored, and performance considerations when constructing an MTBDD.

6 CONCLUSION

We designed and implemented an efficient method to detect equivalence between OpenFlow 1.3 rulesets, the code for which we have released to assist the research community [1].

We detailed how to flatten a multi-table pipeline to an equivalent single-table representation which is easier to work with. We showed that equivalences can exist in actions and present a method of resolving these to a canonical form. Finally, we showed that an MTBDD provides a suitable canonical form to map packet-space to forwarding behaviour and equivalence checking.

Our technique provides a viable option to check the equivalence of OpenFlow 1.3 rulesets which, as far as we know, is the first implementation of SDN ruleset equivalence checking. Future directions for this research include an investigation into incremental optimisations to enable real-time checking of rulesets, optimising BDD node and build order, and developing equivalence checking implementations for other SDN standards such as OpenFlow 1.5 and P4. Our equivalence checking could be used in future work transforming rulesets, be it for optimising power efficiency, targeting new hardware, or reducing table size to improve networks.

REFERENCES

[1] [n. d.]. Equivalence checking implementation [Source Code]. https://github.com/wandsdn/ofequivalence

[2] Sheldon B. Akers. 1978. Binary decision diagrams. *IEEE Trans. Comput.* 6 (1978), 509–516.

[3] David A Applegate, Gruia Calinescu, David S Johnson, Howard Karloff, Katrina Ligett, and Jia Wang. 2007. Compressing rectilinear pictures and minimizing access control lists. In *Proc. 18th annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 1066–1075.

[4] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. 2016. SNAP: Stateful network-wide abstractions for packet processing. In *Proc. 2016 ACM SIGCOMM Conf.* 29–43.

[5] Josh Bailey and Stephen Stuart. 2016. Faucet: Deploying SDN in the enterprise. *Queue* 14, 5 (2016).

[6] Beate Bollig and Ingo Wegener. 1996. Improving the variable ordering of OBDDs is NP-complete. *IEEE Trans. Comput.* 45, 9 (1996), 993–1002.

[7] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.

[8] Karl S Brace, Richard L Rudell, and Randal E Bryant. 1990. Efficient implementation of a BDD package. In *Design Automation Conf., 1990. Proc., 27th ACM/IEEE*. 40–45.

[9] Randal E Bryant. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.* 100, 8 (1986), 677–691.

[10] Edmund M. Clarke, Masahiro Fujita, Patrick C. McGeer, K. McMillan, J.C.-Y. Yang, and X. Zhao. 1993. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Int. Workshop on Logic Synthesis* (1993).

[11] Edmund M Clarke, Kenneth L McMillan, Xudong Zhao, Masahiro Fujita, and Jerry Yang. 1993. Spectral transforms for large boolean functions with applications to technology mapping. In *Proc. 30th int. Design Automation Conf.* 54–60.

[12] Scott Hazelhurst, Anton Fatti, and Andrew Henwood. 1998. Binary decision diagram representations of firewall and router access lists. *Department of Computer Science, University of the Witwatersrand, Tech. Rep* (1998).

[13] Takeru Inoue, Toru Mano, Kimihiro Mizutani, Shin-ichi Minato, and Osamu Akashi. 2018. Fast packet classification algorithm for network-wide forwarding behaviors. *Computer Communications* 116 (2018), 101–117.

[14] Kalapriya Kannan and Subhasis Banerjee. 2013. Compact TCAM: Flow entry compaction in TCAM for power aware SDN. In *Int. Conf. on Distributed Computing and Networking*. Springer, 439–444.

[15] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *NSDI*, Vol. 12. 113–126.

[16] Donald E. Knuth. 2009. *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams*. Addison-Wesley Professional.

[17] Chang-Yeong Lee. 1959. Representation of Switching Circuits by Binary-Decision Programs. *Bell Labs Technical Journal* 38, 4 (1959), 985–999.

[18] David Meyer. 2001. University of Oregon route views archive project. https://routeviews.org/

[19] Open Networking Foundation. 2015. OpenFlow Switch Specification - Version 1.3.5. Retrieved September 8, 2015 from https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.3.5.pdf

[20] osrg. [n. d.]. Ryu SDN Framework. https://osrg.github.io/ryu/

[21] Heng Pan, Hongtao Guan, Junjie Liu, Wanfu Ding, Chengyong Lin, and Gaogang Xie. 2013. The FlowAdapter: Enable flexible multi-table processing on legacy hardware. In *Proc. 2nd ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 85–90.

[22] Heng Pan, Gaogang Xie, Zhenyu Li, Peng He, and Laurent Mathy. 2017. FlowConvertor: Enabling portability of SDN applications. In *IEEE INFOCOM 2017 - Conf. on Computer Communications*. 1–9.

[23] Amit Prakash and Adnan Aziz. 2001. OC-3072 packet classification using BDDs and pipelined SRAMs. In *Hot Interconnects 9, 2001*. IEEE, 15–20.

[24] Steffen Smolka, Spiridon Eliopoulos, Nate Foster, and Arjun Guha. 2015. A fast compiler for NetKAT. *ACM SIGPLAN Notices* 50, 9 (2015), 328–341.

[25] Fabio Somenzi. 2015. CUDD: CU decision diagram package release 3.0. 0. (2015).

[26] WAND. 2018. Redcables SDN Network @ WAND, Waikato University. Retrieved March 12, 2018 from https://redcables.wand.nz/

[27] Hongkun Yang and Simon S. Lam. 2013. Real-Time Verification of Network Properties Using Atomic Predicates. *IEEE/ACM Transactions on Networking* 24 (2013), 887–900.

[28] Liang Yang, Bryan Ng, Winston KG Seah, and Lindsay Groves. 2017. Equivalent forwarding set evaluation in software defined networking. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. 576–579.

[29] Lihua Yuan, Hao Chen, Jianning Mai, Chen-Nee Chuah, Zhendong Su, and Prasant Mohapatra. 2006. Fireman: A toolkit for firewall modeling and analysis. In *2006 IEEE Symposium on Security and Privacy*.