Towards Transforming OpenFlow Rulesets to Fit Fixed-Function Pipelines

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

OpenFlow feature support differs between devices due to device-specific hardware constraints. OpenFlow places the burden of addressing these differences on the controller, which increases development cost and restricts device interoperability. This paper investigates reducing this burden by algorithmically transforming an existing ruleset to fit an incompatible fixed-function pipeline to improve device interoperability. Existing rule-fitting schemes in the literature require metadata to link rules between tables into a path through the pipeline, but not all pipelines support metadata. We developed a novel approach that does not rely on any particular pipeline features, like metadata, and considers the pipeline's constraints, including both the matches and actions available. This paper presents our implementation, including ruleset preprocessing techniques, methods of transforming rules, and how we use a partially constrained boolean satisfiability problem to select from these transformations and build the final ruleset. While future work remains towards real-world deployment, our approach demonstrates fitting rulesets to fixed-function pipelines without metadata is feasible, and our techniques to reduce the size of the problem are beneficial.

CCS CONCEPTS

• **Networks** → **Programming interfaces**; *Programmable networks*;
• **Theory of computation** → *Constraint and logic programming*.

1 INTRODUCTION

Fixed-function pipelines remain common in industry because all major network vendors continue to maintain a range of network hardware based on both fixed-function and programmable pipelines [16, 25]. To program a fixed-function pipeline, an OpenFlow controller must be aware of the layout and function of tables in the pipeline because each table is specialized. For example, Figure 1 shows a typical fixed-function switching table, which can only match the Ethernet destination and the VLAN of a packet and apply a forwarding destination. Currently, most methods of supporting fixed-function pipelines require a developer to write code to target every new pipeline, which is time-consuming and error-prone. In state-of-the-art production controllers, developers write device drivers to transform high-level objectives from applications into device-specific rules [26]. Researchers have suggested other approaches including software and hardware developers agreeing on a common virtual pipeline between controller and switch [9, 28] which requires the hardware manufacturer write the device driver; the development of high-level languages which compile to low-level languages including OpenFlow [21, 23, 27], but not specific devices; and algorithmic methods of converting existing rulesets to fit specific device pipelines [18, 19].

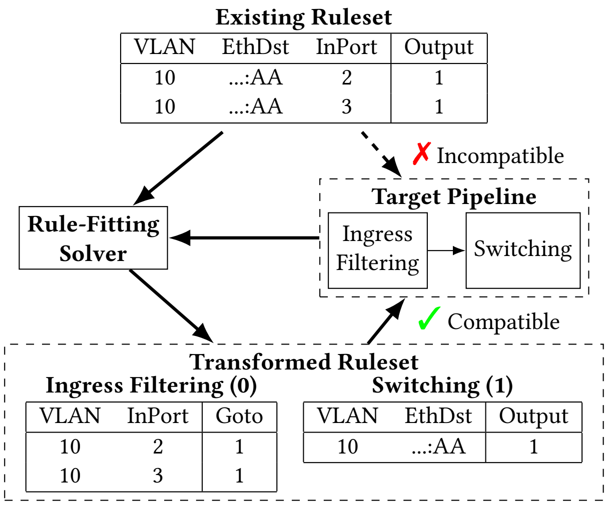


Figure 1: The rule-fitting problem. An OpenFlow controller cannot directly install an existing ruleset if it does not fit the target pipeline. The rule-fitting solver transforms the ruleset to fit the target pipeline.

Our research expands on that last category, the *rule-fitting approach*. The rule-fitting approach does not require modification to either the OpenFlow application or switch. It thus allows a network operator to deploy an otherwise incompatible application to a switch easily. Figure 1 shows, by example, the rule-fitting problem. The rule-fitting solver has split the match and action components of each existing rule between the two tables in the target pipeline in a way that preserves the original forwarding behavior. Previous approaches to the problem, such as FlowAdapter [18] and FlowConvertor [19], create paths through the target pipeline linked using metadata and put all actions in the final rule of this path. This approach fails in pipelines that do not support metadata, or support a restricted set of actions for rules in the final table.

We contribute a novel rule-fitting approach which we designed and implemented to target complex fixed-function pipelines that

SOSR '20, March 3, 2020, San Jose, CA, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7101-8/20/03…$15.00
https://doi.org/10.1145/3373360.3380844

lack metadata support and restrict actions. Our high-level design considers the rule-fitting problem in two parts, (1) generating possible transformations of each rule and (2) finding a combination of these transformations without conflicts. Additionally, we present new techniques that reduce the problem size, including a compression method, which reduces the size of the input ruleset, and a means to filter out transformations that are likely to conflict early. We demonstrate our approach can fit rulesets in synthetic scenarios where the target pipeline does not support metadata and restricts the matches and actions available to a rule. We release an open-source implementation to transform OpenFlow 1.3 rulesets [1] and our evaluation artifacts [2].

This paper is structured as follows; §2 introduces pertinent details of OpenFlow, the Broadcom OF-DPA pipeline, and Table Type Patterns (TTPs). §3 presents our high-level algorithm design. §4 details transforming a rule to fit a pipeline and §5 details selecting a valid combination of these transformations. §6 evaluates our technique and discusses real-world limitations. Finally, §7 provides related work, and §8 concludes.

2 BACKGROUND

2.1 OpenFlow 1.3 Forwarding Model

The OpenFlow 1.3 forwarding model presents a multi-table pipeline, in which all packets are processed by rules in the first table and follow a path through subsequent tables specified by the goto-table instructions of matching rules. Each table contains a priority-ordered list of match-action rules, where each rule specifies the set of packets to match and the corresponding actions to apply. A switch finds the highest-priority matching rule in a table and applies its actions. A rule can match multiple values of a header-field by masking which bits of a field the match considers. OpenFlow defines *metadata* as a special header-field associated with a packet to carry information between tables which rules can match and modify. A controller may use metadata to identify the last rule that processed a packet to create paths through the pipeline.

The actions available to an OpenFlow rule include forwarding a packet and modifying its header-fields. A rule can specify actions in three different ways: *apply-actions*, *write-actions*, and indirectly via *groups*. A switch executes apply-actions immediately, and subsequent tables see the modified packet. A switch adds write-actions to an action-set carried with each packet, which the switch executes on encountering a rule without a goto instruction, i.e. the end of the pipeline. An action-set can be overwritten or cleared by subsequent rules. Rather than adding actions directly to a rule, OpenFlow allows multiple rules to reference a group that contains the desired actions. When a switch is executing actions and encounters a group action, it executes the group referenced. OpenFlow groups contain buckets; each bucket is a set of actions which the switch executes on a copy of the current packet.

2.2 The OF-DPA Fixed-Function Pipeline

We introduce the Broadcom OpenFlow Data Plane Abstraction (OF-DPA) pipeline [5] as a real-world example of the constraints that are present in a fixed-function pipeline. Broadcom's OF-DPA is an OpenFlow 1.3 interface for programming their switching chips. Broadcom's switching chips are popular merchant silicon, with an

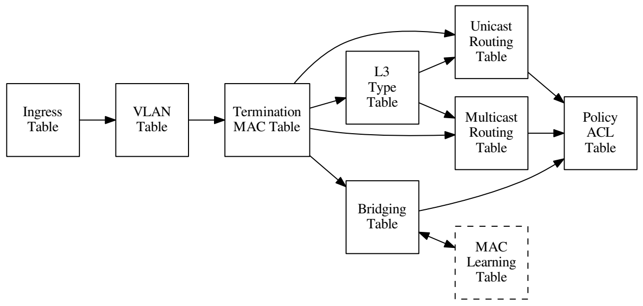


Figure 2: Nine of the 33 tables that OF-DPA 2.0 exposes; these support bridging and routing.

80% market share [12], which many vendors use, including Edge-Core, Quanta, Pica8, Dell, Cisco, and HPE [16, 25]. The OF-DPA pipeline presents specialized tables to efficiently perform fundamental network operations, including switching and routing. The OF-DPA pipeline contains one general table, the policy ACL table, which has the most well-rounded OpenFlow support compared to all other tables in the pipeline. However, the ACL table has a limited rule capacity, due to supporting wide maskable matches, an expensive combination to support in hardware [13, 20].

The OF-DPA pipeline has five key features that present challenges to rule-fitting algorithms.

2.2.1 Restricted Matches and Actions. The tables in fixed-function pipelines are designed to support network functions as cheaply as possible, by using specialized tables that only support narrow exact matches and a subset of actions. Figure 2 shows the OF-DPA 2.0 bridging and routing pipeline [6]. For example, the termination MAC table matches the Ethernet destination of a packet to determine if it should be routed, so that the routing tables do not match the Ethernet destination and, therefore, have a narrower match. Correctly splitting forwarding logic across specialized tables is difficult due to the limits these tables impose on rules.

2.2.2 Prescribed Pipeline. OF-DPA prescribes that rules must always goto a specific table next. For example, a rule-fitting algorithm cannot install a routing rule in the routing table and end pipeline processing, instead the rule must direct packets to the ACL table. This is challenging as a rule-fitting algorithm must ensure that the ACL rules do not unintentionally override this routing decision.

2.2.3 No Metadata Support. OF-DPA does not support OpenFlow metadata. This adds difficulty to the rule-fitting problem as it limits the ability to carry context with a packet through the pipeline, such as the last rule matched.

2.2.4 Overwrites the Action-Set. As Figure 2 shows, the policy ACL table is at the end of the OF-DPA pipeline; after the Routing and Bridging tables that support forwarding decisions. In this design, a controller must add rules to the forwarding tables with a forwarding decision as a write-action, regardless of whether the controller wants to forward all matching packets. This forwarding decision is carried in each packet's action-set. To apply policy, the controller must install a rule in the ACL table that clears or overwrites the forwarding decision in the corresponding packet's action-set. This complicates rule-fitting. Consider a policy rule which drops packets; the rule-fitting solver must first install rules with placeholder forwarding actions in the forwarding tables, and then install a rule in the ACL table to clear the action-set and drop the packets.

2.2.5 No Direct Output Actions. The OF-DPA pipeline does not support output actions in the write-actions or apply-actions of a rule. Instead, to output a packet, a rule must include a group action that, in turn, contains the output action. This requires a rule-fitting solver to interpret groups and move output actions into groups.

2.3 Table Type Patterns

A Table Type Pattern (TTP) [9] is a machine and human-readable description of a logical OpenFlow pipeline, most often encoded in JSON. A TTP describes the types of rules the pipeline supports in a table, the header-fields a rule can match, and the instructions and actions a rule can apply. Additionally, a TTP lists built-in rules which describe the default behavior of a pipeline, such as default table-miss behavior. A TTP also describes the types of groups the pipeline supports, and the actions available to these groups.

A TTP comprehensively describes the requirements of a pipeline. Examples include the match and actions a rule requires, the maskability of a match (exact only, wildcard, prefix, or arbitrarily maskable), valid next tables, and restrictions on the value of a match or action. Examples of a restricted value include an output action only being able to output to the controller, or an Ethernet match that must match a multicast address.

Adoption of the TTP standard is limited; few vendors publicly provide TTP representations of their pipelines. However, the OF-DPA pipeline includes a TTP. We created our own TTP library and tools for this research because there were no existing tools available to verify that a rule fits a TTP or to fit a rule into a TTP.

3 HIGH-LEVEL ARCHITECTURE

Figure 3 shows a design overview of our rule-fitting solver. The rule-fitting solver takes two inputs: (1) a description of forwarding behavior as an OpenFlow 1.3 ruleset [10] and (2) a description of the target pipeline as a TTP [9]. The rule-fitting solver returns an OpenFlow 1.3 ruleset that is compatible with the target pipeline and has forwarding equivalent to the input ruleset. The solver has two main stages, where each stage is a separate problem with unique challenges. The first stage transforms individual rules into the target pipeline, while the second stage searches for a combination of these transformed rules with the desired forwarding.

In the first stage, ruleset preprocessing converts the input ruleset to a single-table (§4.1.1), removes unreachable rules (§4.1.2), and compresses the ruleset into a smaller set of representative rules (§4.1.3) to simplify the rule-fitting problem. Using this preprocessed ruleset, the solver generates a comprehensive set of transformations for each input rule with the same isolated forwarding behavior in the target pipeline. A transformation maps an input rule to *placements*, aka rules, in the target pipeline. This step addresses pipeline complexities, such as moving actions between apply-actions, write-actions, and groups and adding placeholder actions to traverse the pipeline (§2.2.4).

The second stage of the solver (§5) builds the final ruleset by selecting one transformation to represent each input rule, which is

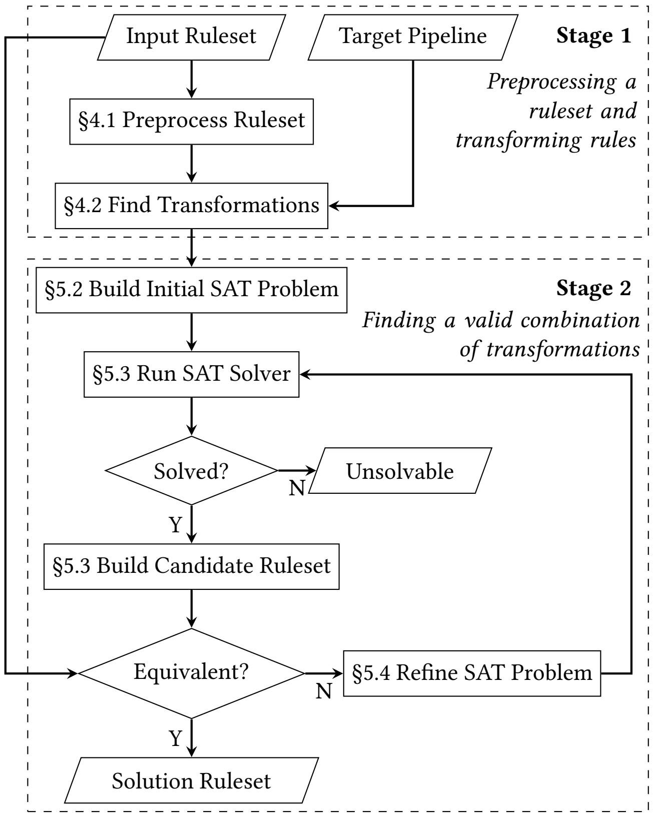

Input Ruleset
Target Pipeline
Stage 1
Preprocessing a ruleset and transforming rules
§4.1 Preprocess Ruleset
§4.2 Find Transformations
§5.2 Build Initial SAT Problem
Stage 2
Finding a valid combination of transformations
§5.3 Run SAT Solver
Solved?
N
Unsolvable
Y
§5.3 Build Candidate Ruleset
Equivalent?
N
§5.4 Refine SAT Problem
Y
Solution Ruleset

Figure 3: The design of our rule-fitting solver. The first stage preprocesses the input ruleset and transforms each input rule, in isolation, into the target pipeline. The second stage produces a solution ruleset with a combination of these transformations that has the same forwarding behavior as the input ruleset.

difficult as their placements often conflict. Conflicts are caused by *shadowing*: when two placements in the same table match a common set of packets, which prevents these packets from reaching the lower priority rule. A conflict occurs when one placement shadows another placement with different actions and results in incorrect overall forwarding behavior. Existing work uses metadata to avoid such conflicts [11, 18, 19, 27]; however, not all pipelines support metadata (§2.2.3). Note, not every instance of shadowing causes incorrect forwarding behavior, in-fact shadowing is needed in all non-trivial OpenFlow rulesets.

Naïvely checking all combinations of transformations is intractable for non-trivial problems. So the second stage uses a partially constrained boolean satisfiability (SAT) problem to generate combinations likely to have the correct forwarding for further verification. We were unable to fully constrain the SAT problem to return only correct combinations, as our attempts to do so all amounted to checking all combinations. So instead, the second stage first identifies placements that we know will conflict or be redundant and builds an initial SAT problem that disallows these conflicts and redundancies (§5.2). Next, it runs the SAT solver, which returns either a combination of transformations or that the problem is unsolvable. From these transformations, the rule-fitting solver builds and verifies the candidate ruleset for equivalence with the input ruleset (§5). If equivalent, the candidate ruleset is a solution to the rule-fitting problem and is returned. Otherwise, our method adds additional constraints to the SAT problem by analyzing forwarding conflicts (§5.4), before rerunning the SAT solver.

4 PREPROCESSING RULESETS AND TRANSFORMING RULES

This section details the first stage of the rule-fitting solver, transforming OpenFlow rules. Transforming rules includes preprocessing the input ruleset to reduce the complexity of the problem (§4.1), and transforming each rule in isolation into placements (rules) in the target pipeline (§4.2) with equivalent forwarding behavior. The placements found in this phase of the solver are used by the second stage (§5), which finds a combination of these placements that do not conflict with each other.

4.1 Ruleset Preprocessing

Before finding transformations for rules, the solver preprocesses the ruleset to simplify the problem. The solver first converts the ruleset to a single table, then removes unreachable rules, and finally compresses the ruleset.

4.1.1 Conversion to a Single-Table. To simplify the problem, the solver first converts the multi-table input ruleset to an equivalent single-table ruleset. A single-table ruleset represents each path through the input ruleset as a single rule containing the entire forwarding decision. Single-table input simplifies the rule-fitting problem because the solver only needs to consider splitting rules across tables, compared to a multi-table input where the solver must consider combinations of both merging and splitting rules.

There is no additional overhead to convert the ruleset to a single-table because we use the library from [22] to verify the equivalence of our result, which already requires conversion to a single-table. A single-table ruleset typically has more rules than the original multi-table ruleset as it is the Cartesian product of these tables, so the solver has more rules to fit. However, in practice, we find the ruleset compression (§4.1.3) preprocessing step reduces the number of rules to alleviate this issue.

4.1.2 Removing Unreachable Rules. At first glance, every rule in the input ruleset has a purpose, and therefore, the solver must represent every rule in the output ruleset. However, this is an incorrect assumption because rulesets may contain unreachable rules. Conversion to a single-table commonly results in unreachable rules with action combinations that the solver cannot place. Besides making a problem solvable, eliminating unreachable rules also improves the performance of the solver because there are fewer rules to consider. A rule is unreachable when higher-priority rules prevent any packets from reaching the rule, i.e. the rule is fully-shadowed.

To find unreachable rules, the solver considers the single-table representation of the input ruleset in descending priority order and adds the packets each rule matches to a set representing the packets matched by all higher-priority rules. This set begins empty. The solver checks if higher-priority rules fully-shadow the rule by calculating the union of the set built so far with the set of packets the rule matches. If the resulting set is unchanged, this rule is unreachable, and the solver removes the rule; otherwise, the solver keeps the rule. Our implementation uses Binary Decision Diagrams (BDDs) to represent sets of packets efficiently for this calculation [22].

4.1.3 Ruleset Compression. Ruleset compression is a novel technique we developed to reduce the size of the input ruleset to the rule-fitting problem. Most non-trivial rulesets will contain multiple rules that perform the same network function, for example, a ruleset may contain a forwarding rule for every host learned. Intuitively, such rules will be similar to each other, as they come from the same code path in the controller, and the rule-fitting solver should fit them to the same place in the target pipeline. The idea behind compression is to select one representative rule to replace a group of similar rules to reduce the size of the ruleset.

The compression algorithm first creates coarse groups of 'similar' rules which have the: (1) same match mask (i.e. match the same bits of the packet header), (2) same action types, and (3) same priority. Next, the algorithm ensures these groups represent the relationships between the rules they contain by ensuring all rules in a group have the same inter-group dependencies. A rule holds a dependency with another rule when their matches overlap (shadow) and holds an inter-group dependency with the corresponding group. The compression algorithm iteratively splits groups where rules hold different inter-group dependencies until all rules within the same group hold the same inter-group dependencies. Finally, the algorithm builds the compressed ruleset by considering each group in priority ascending order and picking one rule to represent each, which, for each inter-group dependency, holds a dependency with the corresponding rule picked so far (if any). This process is similar to the abstraction refinement in Bonsai [4].

A compressed ruleset contains fewer rules than the original, so it is faster for the rule-fitting solver to fit, but it has different forwarding behavior. Therefore, any solution must be mapped back to the original ruleset. The solver maps the solution back to the original ruleset by applying the solution found to fit each representative rule back to the original rules in the same group.

4.2 Finding Rule Transformations

Given an input ruleset as a single-table and a TTP, the solver finds ways to transform each input rule into the target pipeline. A transformation is a mapping from one input rule to one or more placements in the target pipeline. A transformation's placements must have forwarding equivalent to the input rule for the packets that the input rule matches. The next stage of the solver is responsible for selecting a combination of transformations, one per input rule, which do not conflict with each other and result in the correct overall forwarding behavior.

4.2.1 Generating Rule Placements. Our algorithm begins by generating all possible placements of an input rule into the target TTP on a per-table basis. These placements are building blocks that we use next to build transformations. Our algorithm generates a placement for all combinations of the input rule's matches and actions that a table supports. There are two resulting types of placements: full and partial. A full placement has the same match and actions as the input rule; otherwise, it is a partial placement. A full placement will match the same packets and apply the same forwarding as the original rule. However, partial placements, if missing a match, will match more packets than the original, and if missing an action, will apply different forwarding behavior.

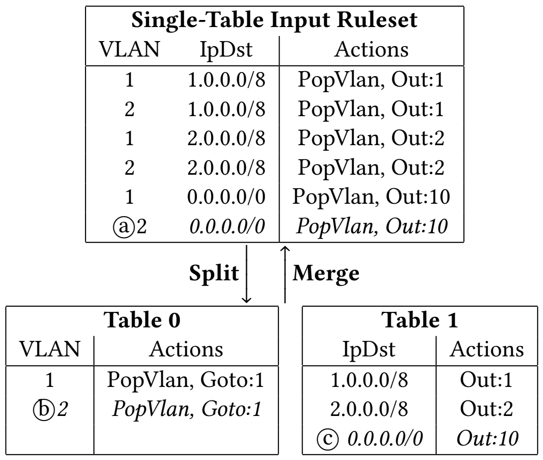
Beyond trying to place actions precisely as they appear in the input rule, we consider variations on these actions. Because some pipelines only support actions in one of apply-actions, write-actions, or groups (§2.2.4 & §2.2.5), we allow actions to move between apply-actions, write-actions, and groups when generating placements. Because some pipelines require placements which clear the action-set (§2.2.4), we generate variations of placements both with and without the clear-actions instruction. To explore all paths through the pipeline (§2.2.2), we generate variations of placements with all tables they can go to next, including no next table.

4.2.2 Placement Priorities. The relative priorities of rules determine which rule the switch chooses to process a packet. The preprocessed single-table input, for which the solver finds transformations, is priority-ordered and contains shadowing. The placements the solver generates retain the same priority as the input rule. §4.2.6 provides a mechanism to lower these priorities when a transformation builds a path through an unsuited table, often via a table-miss rule which does not modify packets, while still allowing more specific rules which use the functionality of that table to override the table-miss rule.

4.2.3 A Direct Transformation. A direct transformation maps an input rule to one placement in a table with the same matches and actions. The solver generates a direct transformation for each full placement of an input rule. The solver can find multiple direct placements in the same table due to the action variations.

Direct placements find placements for policy rules, such as dropping a traffic class, because they can generate a placement in an ACL table independent of how packets reach that table. The next stage of the solver determines a combination of transformations that directs the correct traffic to reach such a placement.

4.2.4 A Split Transformation. The method in §4.1.1 converted the original ruleset to a single-table by merging rules spread across different tables into one rule. A split transformation is the opposite of this merge operation; it splits an input rule into multiple placements which form a path through the target pipeline. Figure 4 shows how rule ⓐ can be split into placements ⓑ and ⓒ. The individual placements of a split transformation will often match a broader set of packets than the original rule and apply only a portion of the original actions. Consider the split transformation shown in Figure 4, ⓐ is split into ⓑ and ⓒ, both of which match a broader set of packets than the original rule. Therefore, placement ⓒ applies the output action from ⓐ to all IP packets; however, ⓐ only matched packets that were both IP packets and had a VLAN of 2. The solver's next stage determines a combination of transformations that avoids these broad placements conflicting with other placements.



Single-Table Input Ruleset

VLAN	IpDst	Actions
1	1.0.0.0/8	PopVlan, Out:1
2	1.0.0.0/8	PopVlan, Out:1
1	2.0.0.0/8	PopVlan, Out:2
2	2.0.0.0/8	PopVlan, Out:2
1	0.0.0.0/0	PopVlan, Out:10
ⓐ2	*0.0.0.0/0*	*PopVlan, Out:10*

Table 0

VLAN	Actions
1	PopVlan, Goto:1
ⓑ*2*	*PopVlan, Goto:1*

Table 1

IpDst	Actions
1.0.0.0/8	Out:1
2.0.0.0/8	Out:2
ⓒ *0.0.0.0/0*	*Out:10*

Figure 4: A demonstration of splitting a single-table input ruleset to fit a multi-table pipeline. A split transformation fits ⓐ into the target pipeline as two placements ⓑ and ⓒ. Whereas, joining ⓑ and ⓒ to form ⓐ is the merge operation preprocessing uses to create a single-table.

Placed Portion		Unplaced Portion		
VLAN	Actions	VLAN	IpDst	Actions
2	PopVlan, Goto:1	—	0.0.0.0/0	Out:10
2	Goto:1	—	0.0.0.0/0	PopVlan, Out:10
2	PopVlan	—	0.0.0.0/0	Out:10
2	—	—	0.0.0.0/0	PopVlan, Out:10
—	PopVlan, Goto:1	2	0.0.0.0/0	Out:10
—	Goto:1	2	0.0.0.0/0	PopVlan, Out:10
—	PopVlan	2	0.0.0.0/0	Out:10
—	—	2	0.0.0.0/0	PopVlan, Out:10

Figure 5: Partial placements of rule ⓐ into Figure 4 *Table 0*. There are no full placements because all placements have unplaced portions. Rule ⓐ matches VLAN:2, IpDst:0.0.0.0/0 and applies the actions PopVlan, Out:10. Where *Table 0* can optionally match VLAN, apply a PopVlan action, and goto-table 1. The solver generates partial placements with Goto:1 as a variation it tries.

The solver generates split transformations using the partial placements of the input rule. Figure 5 shows an example of partial placements of ⓐ into Figure 4 *Table 0*. The solver finds all valid paths through these partial placements, starting from a placement in the first table, following each placement's goto instruction, and selecting one rule in each table following. If a path applies the same forwarding as the original rule, for the packets the original rule matched, we say the path has the same *isolated forwarding* and generate a split transformation.

4.2.5 Filtering Split Transformations. The number of paths that the solver checks when generating a split transformation is the product of the partial placements it finds for each table. The number of paths to check is large for long pipelines. Reducing this number without excluding valid solutions benefits solver performance.

Input Ruleset

	IpDst	TcpDst	Actions
ⓓ	192.168.1.0/24	22	Drop
ⓔ	192.168.1.0/24	–	Out:1

Target Pipeline Requirements

Routing Table (0): Must match an IpDst subnet. Actions **can** add an output to the action-set; **must** goto ACL table.

ACL Table (1): Can include any arbitrary match. Actions **can** clear the action-set.

Split Transformations (per-rule)

	Routing Table (0)		ACL Table (1)		
	IpDst	Apply-Actions	IpDst	TcpDst	Clear-Actions
ⓓ	192.168.1.0/24	Goto:1	192.168.1.0/24	22	Yes
ⓔ	192.168.1.0/24	Out:1, Goto:1	192.168.1.0/24	–	No

Figure 6: An example where ⓓ's split transformation placement needs a placeholder action to avoid conflicting with ⓔ's placement in the routing table. If the solver was to install both transformations, ⓓ's placement in the routing table takes priority over ⓔ's placement. Therefore the default forwarding decision in ⓔ is lost, and forwarding is incorrect. However, if ⓓ's placement in the routing table was replaced with ⓔ's placement, then the forwarding is correct.

Instead of building paths with all possible partial placements, the solver only builds paths using the partial placement with the most specific match for each unique set of actions per table. Consider the partial placements shown in Figure 5. Because the VLAN match is optional in the target pipeline, each placement has a variation with and without the VLAN match. The variations without the VLAN will match packets with any VLAN, and are more likely to conflict with other placements. Therefore, the solver only builds paths with the first four partial placements in Figure 5, which include the VLAN match.

More precisely, the solver filters partial placements before generating split placement paths. The solver removes any partial placements which have the exact same actions as another and where the match is a superset of the other placement's match. The superset constraint ensures the solver does not remove placements where the match fields are orthogonal.

4.2.6 Adding Additional Placeholder Actions. As discussed in §2.2.4, some fixed-function pipelines use the packet's action-set to store the default forwarding decision and require a rule in a later table to override this decision. Figure 6 shows a simplified portion of the OF-DPA pipeline, which demonstrates why the solver must generate split transformations in which placements include a placeholder action. Consider the single-table input ruleset shown; it contains two rules: ⓓ a policy rule, and ⓔ a forwarding rule. The policy rule shadows the forwarding rule for TCP destination port 22 and drops these packets. Because the routing table can only match the IPv4 destination, ⓓ has no forwarding decision (output action), and split transformations maintain their original priority; the placement of ⓓ completely shadows the placement of ⓔ in the routing table and drops all packets. The next stage of the solver picks one transformation per input rule. Thus, if this stage of the solver only found the two transformations listed, the next stage must pick both, which results in the wrong overall forwarding behavior.

The solution is to substitute ⓓ's placement in the routing table with ⓔ's placement. A substitution is acceptable if it retains the same isolated forwarding as the input rule. Note that in this simple case, a direct transformation of ⓐ in the ACL table also solves this problem.

In order to avoid such conflicts, the rule-fitting solver generates new split transformations by substituting a placement in a split transformation with a placement from another input rule. A substitution is valid if the placement: (1) is in the same table, (2) goes to the same table, (3) has the same match as the original placement, and (4) results in the same isolated forwarding as the input rule when substituted into the existing split transformation. This process retains the original, and normally lower, priority of the substituted rule, and will substitute placements which only differ by priority. Lowering the priority in this way makes intuitive sense as ⓓ can only install the IPv4 part of its match in the routing table, so it should do so at a lower-priority to allow a placement with a more specific match to override ⓓ. That is, lowering the priority reverses the process of merging a ruleset into a single-table, which adds the rules' priorities along a path to form a single rule.

Our solver only considers deviations of one placement changed from the original split transformation. Doing otherwise would increase the space the solver searches for possible solutions, but unfortunately, results in a substantial expansion in compute time. Future research is required to find more efficient ways of generating and representing these equivalent variations of split transformations.

5 FINDING A VALID COMBINATION OF TRANSFORMATIONS

The first stage of our rule-fitting solver outputs a list of possible transformations for each input rule. In isolation, each transformation applies the correct forwarding. However, when our solver combines these transformations, their placements can conflict. For example, when a placement shadows another or when a placement is in an unreachable table. The main challenge is that for any non-trivial input ruleset, naïvely verifying all combinations is intractable. Our method uses a SAT problem to return the combinations of transformations where valid solutions are most likely.

§5.1 introduces the SAT problem and provides an overview of our implementation. §5.2 details how our solver creates an initial set of SAT constraints to remove combinations that are very unlikely to result in the correct forwarding. The rule-fitting solver gives these constraints to the SAT solver, which returns a combination of transformations. §5.3 describes how our solver builds the corresponding candidate ruleset and checks the forwarding equivalence against the input ruleset. If a candidate is not equivalent, §5.4 describes how our solver adds constraints for the specific placements with conflicting forwarding, then reruns the SAT solver.

5.1 Boolean Satisfiability (SAT)

Consider the boolean expression $(A \land B) \lor C$. When $A = T$, $B = T$, and $C = F$, this expression is said to be *satisfied* because it evaluates to true. The SAT problem asks if it is possible to satisfy a boolean

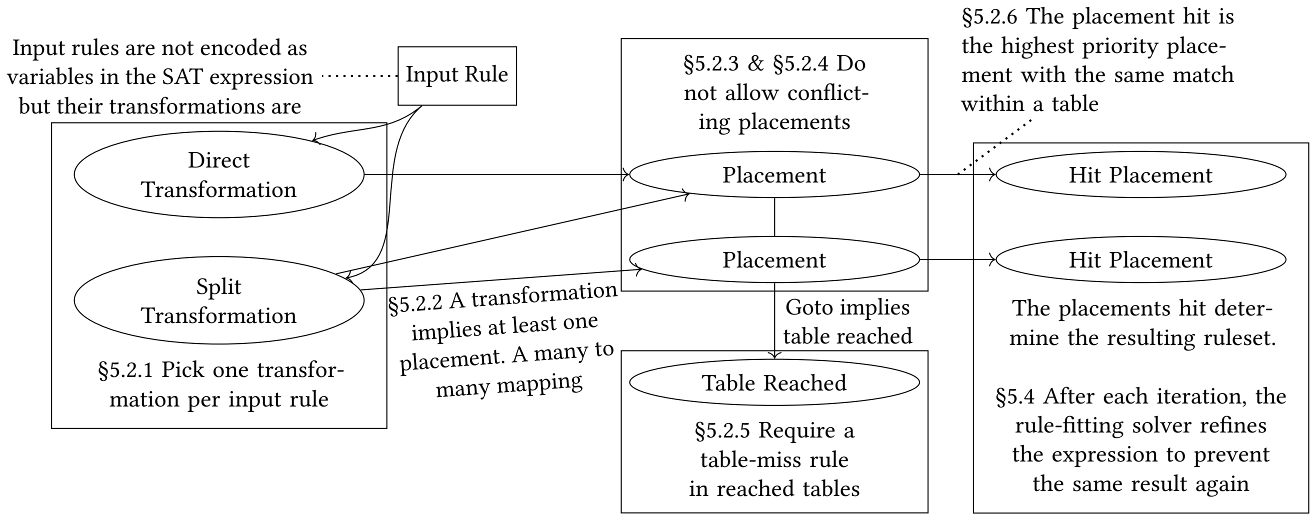

Input rules are not encoded as variables in the SAT expression but their transformations are
Input Rule
Direct Transformation
Split Transformation
§5.2.1 Pick one transformation per input rule
§5.2.2 A transformation implies at least one placement. A many to many mapping
§5.2.3 & §5.2.4 Do not allow conflicting placements
Placement
Placement
Goto implies table reached
Table Reached
§5.2.5 Require a table-miss rule in reached tables
§5.2.6 The placement hit is the highest priority placement with the same match within a table
Hit Placement
Hit Placement
The placements hit determine the resulting ruleset.
§5.4 After each iteration, the rule-fitting solver refines the expression to prevent the same result again

Figure 7: An overview of the SAT expression the rule-fitting solver constructs. Elliptical nodes represent boolean variables. Labeled edges describe the relationship between variables. Groups describe the constraints between the enclosed variables. Input rules are not variables in the SAT expression, but their transformations from the first stage of the rule-fitting solver are.

expression; therefore, to determine if an assignment of the variables A, B, and C exists where the expression evaluates to true.

Our rule-fitting solver uses the SATisPy [15] library to build the boolean expression in the de-facto standard DIMACS [7] format and MiniSat 2 [8] as the SAT solver. Figure 7 shows an overview of the SAT expression our rule-fitting solver builds. We designed the SAT constraints to reduce the search space by preventing candidate solutions that are duplicates or extremely likely to have incorrect forwarding behavior. Direct and split transformations from the first stage map directly to SAT variables. The following sections logically define how our method derives all other variables and constraints. The rule-fitting solver generates the majority of the SAT clauses once, as initial constraints. The exception to this is the refinements that the rule-fitting solver adds to each iteration (§5.4).

5.2 The Initial SAT Expression

This section details the initial constraints of the rule-fitting problem and how our method encodes these into a SAT expression. Appendix A lists the boolean and set operations we use in the following sections to define clauses. The SAT solver returns solutions which satisfy all clauses, i.e. clauses are logically AND'd together. The order of this section follows Figure 7 from left to right.

5.2.1 Include a Transformation of Every Rule. After preprocessing, the input ruleset does not contain any redundant rules, which means a valid solution to the rule-fitting problem needs to represent every rule. Therefore, the rule-fitting solver needs to select a transformation of every rule to create an equivalent ruleset. It is generally wrong to pick more than one transformation. Placements from two or more transformations in different tables are likely to apply the wrong forwarding by incorrectly applying the same action twice. Further, such placements in the same table are redundant as a packet can only match one.

$$\forall r \in \text{ruleset} : \qquad \textsc{onehot}\big(\text{transformations of } r\big) \quad (1)$$

Therefore, the solver adds a constraint to pick exactly one transformation for each rule. For every input rule, r, Equation (1) creates a clause using ONEHOT to ensure all solutions contain precisely one transformation, either direct or split.

5.2.2 Placement Variables. A transformation maps an input rule to concrete placements in the target pipeline. In order to add constraints to these placements, we map transformations to their corresponding placements. It is common for multiple transformations to map to the same placement, such as a split transformation using a table-miss rule that passes packets through a table unaltered.

$$\begin{aligned} &\forall t \in \text{transformations} : \\ &\quad \forall p \in \text{placements of } t : \qquad t \rightarrow p \end{aligned} \quad (2)$$

Equation (2) links, using implication, each transformation t, to each corresponding placement, p. A placement p represents a unique rule in the final ruleset with the same match, priority, table, instructions, and actions.

$$\begin{aligned} &\forall p \in \text{placements} : \\ &p \rightarrow \bigvee \{t \in \text{transformations } \textbf{where } p \in \text{placements of } t\} \end{aligned} \quad (3)$$

Equation (2) is not sufficient by itself, as a placement can be true without any corresponding transformations selected. Equation (3) adds a clause for each placement variable, p, to ensure it is true only when at least one corresponding transformation, t, is true.

5.2.3 Disallow Same-Priority Conflicting Placements. OpenFlow rules at the same priority with overlapping matches but with different instructions (which include actions) have undefined forwarding behavior [10] as it is unclear which takes priority.

$$\begin{aligned}
&\forall p_1 \in \text{placements}, \forall p_2 \in \text{placements } \textbf{where}\\
&\qquad p_1 \neq p_2 \land\\
&\qquad \textsc{priority}(p_1) = \textsc{priority}(p_2) \land\\
&\qquad \textsc{table}(p_1) = \textsc{table}(p_2) \land\\
&\qquad \textsc{match}(p_1) \cap \textsc{match}(p_2) \neq \emptyset \land\\
&\qquad \textsc{instructions}(p_1) \neq \textsc{instructions}(p_2) :\\
&\qquad\qquad \neg(p_1 \lor p_2) \qquad (4)
\end{aligned}$$

Equation (4) shows the clauses the solver generates to disallow such conflicts. The equation considers all pairs of placements at the same priority with an intersecting match, and disallows any combination with different instructions.

5.2.4 Disallow Placements with Conflicting Instructions. A switch applies only the highest priority matching rule in a table to a packet. Thus if a placement is fully-shadowed by a higher-priority placement, the higher-priority placement must be a valid substitute for the shadowed placement. The first stage of the solver generates a new transformation for all valid substitutions when considering placements with placeholder actions (§4.2.6). Therefore, for all valid substitute placements, the SAT solver will choose another transformation with that placement.

$$\begin{aligned}
&\forall p_{lo} \in \text{placements}, \forall p_{hi} \in \text{placements } \textbf{where}\\
&\qquad \textsc{priority}(p_{lo}) < \textsc{priority}(p_{hi}) \land\\
&\qquad \textsc{table}(p_{lo}) = \textsc{table}(p_{hi}) \land\\
&\qquad \textsc{match}(p_{lo}) \subseteq \textsc{match}(p_{hi}) \land\\
&\qquad \textsc{instructions}(p_{lo}) \neq \textsc{instructions}(p_{hi}) :\\
&\qquad\qquad \neg(p_{lo} \land p_{hi}) \qquad (5)
\end{aligned}$$

Therefore, the rule-fitting solver adds clauses to disallow a combination of placements within the same table with the same match but different instructions, including actions. For all conflicting, fully-shadowed placements, p_{lo}, and the corresponding higher priority rule, p_{hi}, Equation (5) creates a clause to disallow both placements together. As a result, the final SAT expression is only satisfied when all fully-shadowed placements have the same instructions as the placement hit instead.

5.2.5 Require a Table-Miss Rule. In OpenFlow, a table-miss rule is placed at the lowest priority in each table to match all packets not matched by other rules. Rather than rely on default switch behavior, we explicitly install a table-miss in all reachable tables. Requiring an explicit table-miss rule reduces the search space because a ruleset with an explicit table-miss rule with the switch's default table-miss action is equivalent to a ruleset without a table-miss rule, but are different rulesets.

$$\begin{aligned}
&\forall p \in \{\text{placements } \textbf{where } p \text{ has a goto instruction}\} :\\
&\qquad p \longrightarrow tr_x \textbf{ where } tr_x \text{ represents the next table } x\\
&\textbf{and}\\
&\qquad tr_0 \qquad \triangleright \text{ table 0 is always reached} \qquad (6)
\end{aligned}$$

First, Equation (6) maps every placement to the corresponding table it goes to, tr_x. Thus, when true, tr_x represents that packets reach table x. Additionally, because a switch begins processing all packets in the first table, its corresponding variable tr_0 must always be true.

$$\begin{aligned}
&\forall tr_x \in \text{tables-reached} :\\
&tr_x \longrightarrow \bigvee \{p \in \text{table-miss placements } \textbf{where } \textsc{table}(p) = x\} \qquad (7)
\end{aligned}$$

Second, Equation (7) requires that all reachable tables have a table-miss rule.

5.2.6 Hit Placement Variables. A switch only applies the highest priority placement in a table. Therefore, if placement a fully-shadows placement b, placement b does not affect forwarding. Changing only these shadowed placements does not change forwarding, so considering such cases is unproductive. Therefore, we create *hit placement* variables in the SAT problem to track placements that affect forwarding. Hit placements variables uniquely define a candidate solution, which the method in §5.4 ensures the SAT solver does not return again on subsequent iterations.

$$\begin{aligned}
&\forall p_x \in \text{placements} :\\
&\qquad \{p_0, p_1 \ldots p_n\} \in \text{placements } \textbf{where}\\
&\qquad \textsc{match}(p_x) \subseteq \textsc{match}(p_n) \land\\
&\qquad \textsc{table}(p_x) = \textsc{table}(p_n) \land\\
&\qquad \textsc{priority}(p_x) < \textsc{priority}(p_n) :\\
&\qquad\qquad (p_x \land \neg p_0 \land \neg p_1 \land \ldots \land \neg p_n) \leftrightarrow h_x \qquad (8)
\end{aligned}$$

For each placement, p_x, Equation (8) maps the corresponding hit placement variable h_x to be true only when hit. p_x is hit when it is true itself (i.e. selected for the candidate solution) and no higher priority placements that fully-shadow it are true ($\{p_0, p_1 \ldots p_n\}$).

5.2.7 Built-In Rules. Our method maps rules built into a pipeline, as per the TTP, as direct placements that the SAT expression always selects. Because built-in rules cannot be removed, our method in §5.2.4 does not consider built-in rules to have conflicting instructions with other placements, as the only way to override a built-in rule is with a placement with conflicting instructions.

5.3 Verifying the Candidate Ruleset

The rule-fitting solver gives the initial SAT expression to the SAT solver, which either returns a solution or that the boolean expression is unsatisfiable. If unsatisfiable, the rule-fitting problem is unsolvable. Otherwise, the SAT solver returns a solution with all variables assigned to a concrete value, either true or false.

The rule-fitting solver collects all selected transformations and builds the corresponding candidate ruleset. The solver then verifies if the candidate ruleset has equivalent forwarding to the input ruleset using the library from [22]. If forwarding is equivalent, the rule-fitting solver has found a valid solution which it can return. Otherwise, the candidate ruleset was not equivalent, so the rule-fitting solver refines the SAT expression (§5.4) and reruns the SAT solver. The rule-fitting solver repeats this process until it finds a valid solution or determines the problem is unsatisfiable.

Termination (0)	
Match	Action
EthDst	Goto
...:01	1
...:02	1
—	2

Routing (1)		
Match	Write-Actions	
IpDst	Output	Goto
1.0.0.0/8	20	3
10.0.0.0/8	20	3
—	21	3
—	—	3

Switching (2)		
Match	Write-Actions	
EthDst	Output	Goto
...:0a	10	3
...:0b	11	3
...:0c	12	3
—	flood	3

TCP Filtering (3)			
Match		Write-Actions	
TcpDst	EthDst	Clear-Actions	Goto
80	—	yes	—
443	—	yes	—
—	...:01	no	—
—	...:02	no	—
—	—	no	4

Learning (4)		
Match		Apply-Actions
EthSrc	InPort	Output
...:0a	10	—
...:0b	11	—
...:0c	12	—
—	—	controller

Figure 8: The *5-table pipeline* and the corresponding ruleset used in this evaluation. This 5-table simplification of the OF-DPA pipeline retains most OF-DPA complexities, such as write-actions in the routing and switching tables. The TCP filtering table can clear these write-actions. The lowest-priority table-miss rule in each table is built into the pipeline. In addition to what is shown, rules in the routing table rewrite a packet's Ethernet address; otherwise, tables support only the features, including the specific next table, listed.

5.4 Refining the SAT Expression

This section defines the clauses the rule-fitting solver adds to the SAT expression when the last candidate ruleset was invalid, before rerunning the SAT solver.

5.4.1 Ensure a New Solution. As described in §5.2.6, the hit placement variables represent a unique candidate ruleset. Once the SAT solver returns a solution, the rule-fitting solver adds a clause to prevent that same solution, i.e. combination of hit placements, again, shown in Equation (9).

$$\bigvee \left\{ \forall h \in \text{hit-placements} : \begin{array}{ll} \neg h, & \textbf{if } h = True \\ h, & \textbf{if } h = False \end{array} \right\} \quad (9)$$

5.4.2 Disallowing Forwarding Conflicts. Beyond calculating if a candidate ruleset has equivalent forwarding, the verification process also returns the set of packets that observe incorrect forwarding. For these packets, the rule-fitting solver creates a mapping from the input rule which processed these packets to the corresponding incorrect path, i.e. placements, in the candidate solution. At least one placement must be from another input rule and shadows packets from reaching a placement of the input rule. Thus, we add a clause to prevent this conflict again in future solutions, which reduces the overall search space.

$$\begin{aligned} &\forall (r_i, path_r) \in \text{forwarding conflicts :} \\ &\qquad \textbf{Let } t = \text{selected transformation of } r_i \\ &\qquad \textbf{Let } \{h_1 ... h_n\} = \text{hit placements of } path_r \\ &\qquad\qquad \neg(t \wedge h_1 \wedge ... h_n) \quad (10) \end{aligned}$$

Equation (10) prohibits selecting the same transformation t with the same conflicting hit placements $\{h_1...h_n\}$ again. All new solutions must either exclude a conflicting hit placement or select a new transformation of the input rule, r_i.

6 EVALUATION

This section presents an evaluation of our rule-fitting solver. We first evaluate the effectiveness of ruleset compression and our SAT constraints, separately, at bringing the rule-fitting problem down to a tractable size. For this evaluation, we used two synthetic rulesets and pipelines, one of which we based on OF-DPA to capture its complexities. Then we discuss the complexities of real-world rulesets and pipelines and give insight into the weaknesses of our approach.

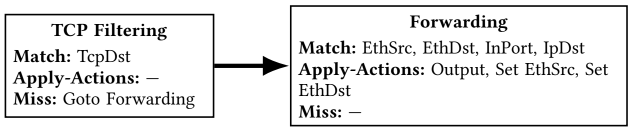

TCP Filtering
Match: TcpDst
Apply-Actions: —
Miss: Goto Forwarding
Forwarding
Match: EthSrc, EthDst, InPort, IpDst
Apply-Actions: Output, Set EthSrc, Set EthDst
Miss: —

Figure 9: The *2-table pipeline*. A simple pipeline compatible with the 5-table pipeline shown in Figure 8. We designed the pipeline to contrast with the 5-table pipeline, such that conversion between pipelines requires the extensive transformation of the ruleset.

6.1 Measurement Methodology

All performance testing was performed on a Ubuntu 16.04 machine, with an Intel i7-4790 @ 3.6Ghz (boost 4.0Ghz), with 8GB of RAM and the Linux 4.15 kernel. Our rule-fitting solver implementation [1] is a single-threaded Python application, which we ran on Python 2.7. We used a script to collect performance results and repeat each test 10 times. Each test was preceded by a warm-up run to load files the test accesses into memory. We report the mean along with the 95% confidence interval (CI). The rule-fitting solver measures its own total run-time and excludes the time to load Python and supporting libraries, which took 0.6s. Internally, the solver uses unordered data structures, and run-to-run may explore different candidate solutions before finding a valid solution.

For this evaluation, we constructed two pipelines and corresponding rulesets. We perform this analysis on these small rulesets, as without all SAT constraints or ruleset compression, the size of the problem grows immensely, and larger problems become intractable.

Figure 8 shows the first pipeline, the **5-table** pipeline, which we based on the OF-DPA bridging and routing pipeline. The main difference from the original OF-DPA pipeline is that we have removed VLAN matches. We retain the other complexities of the pipeline. Figure 8 also shows the ruleset we used to evaluate SAT constraints.

We crafted the other pipeline, the **2-table** pipeline, shown in Figure 9, with a contrasting table layout while still supporting the same forwarding. This pipeline performs all forwarding in the second table using apply-actions, unlike the 5-table pipeline, which uses write-actions spread over multiple tables. To convert rules between these two pipelines, the rule-fitting solver must make significant transformations to the rules. For brevity, we omit the equivalent 2-table ruleset and instead release it along with our evaluation artifacts [2].

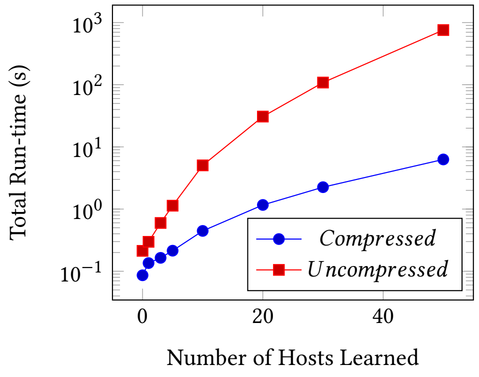


Figure 10: The rule-fitting solver performance comparing compressed vs. uncompressed rulesets when fitting the 5-table ruleset back into the 5-table pipeline. In all cases, ruleset compression significantly reduced the solver run-time.

6.2 Ruleset Compression Performance

To quantify the trade-off between the overhead of compressing (§4.1.3) a ruleset and the resulting speed-up to the rule-fitting solver, we ran an experiment to compare the performance of the rule-fitting solver with and without compression for rulesets of different sizes. The experiment compared the performance of fitting the 5-table ruleset, after conversion to a single-table, back into itself (Figure 8). We configured the solver to find the first valid solution. We changed the size of the ruleset by varying the number of learned hosts. For each host learned, we placed a corresponding rule in both the Switching and Learning table. Figure 8 shows the ruleset with 3 hosts (...:0a, ...:0b, and ...:0c) learned.

Figure 10 plots the result of this experiment; 95% CIs are too small to show, so the run-time is predictable for this experiment. In all cases, we found that ruleset compression was beneficial and reduced the overall solve time. With no hosts learned, solving took 86ms compressed compared to 213ms uncompressed, and with 50 hosts learned, 6.1s compared to 12min 38s.

6.2.1 Compression of Real-world Rulesets. Our synthetic test case is a near best-case scenario. With 50 hosts learned, the uncompressed ruleset as a single-table contained 2,721 rules (due to the Cartesian product expansion of the tables) and compressed down to just 10 rules. We have compared compression on two real-world rulesets. The first ruleset was collected from a switch controlled by Faucet [3] and contained 1,937 rules in the original multi-table pipeline, 3,901 rules as a single-table, and 94 rules once compressed. The second ruleset was collected from a router controller by Faucet and contained 582 rules, 5,281 rules as a single-table, and 902 rules once compressed. In both cases, compression significantly reduced the size of the single-table ruleset; for the switch ruleset, compression removed 97.6% of the rules, and for the router, compression removed 82.9% of the rules.

6.3 Effectiveness of SAT Constraints

This section presents an evaluation of the effectiveness of the SAT constraints described in §5. We configured the solver to return all solutions it could find and limited the solver to consider only 10,000 candidate solutions. Our evaluation started from the least constrained SAT problem, picking one transformation per rule, then cumulatively added the constraints from §5. In order, we begin with: *One Transformation* (§5.2.1), then added, *Placements* (§5.2.2), *Placement Conflicts* (§5.2.3 & §5.2.4), *Table-Miss* (§5.2.5), *Hit Placements* (§5.2.6), and finally *Forwarding Conflicts* (§5.4.2) from the refinement step. With only the *one transformation* constraint, the solver naïvely tries all combinations of transformations; once we added *placements* (and later *hit placements*), these define a unique solution, which the SAT solver will not return again (§5.4.1).

Table 1 shows the performance results when converting the 5-table ruleset to the 2-table pipeline, and Table 2 shows the reverse direction. For both directions, we observed that adding SAT constraints either significantly decreased the total solve time by reducing the number of candidate solutions, or incurred negligible overhead. Table 2 shows without constraints, the solver reached the 10,000 candidate solution limit and took 13.3s, and with all constraints, the solver verifies all 10 candidate solutions in 128ms. Table 1 shows without constraints, the solver verified 16 candidate solutions in 188ms, and with all constraints, the solver verified 4 candidate solutions in 135ms.

To verify that our SAT constraints only remove either invalid or duplicate candidate solutions, we recorded the number of valid solutions the solver found and how many of those were unique. We found that our constraints successfully prevented the SAT solver from returning the same candidate solution to be verified more than once since the number of valid solutions equaled the number of unique solutions when we added all constraints. We found that our SAT constraints have not incorrectly removed valid solutions since the number of unique solutions found remained constant regardless of the constraints, with one exception, the first row of Table 2. The experiment in the first row of Table 2, with only one transformation constraint, reached the candidate solution limit, so the solver only found seven unique solutions rather than ten.

6.4 Real-World Considerations

So far, we have evaluated against our handcrafted rulesets and pipelines where we can be sure that a solution to the rule-fitting problem exists. However, rule-world rulesets and pipelines bring with them a scale and complexity that makes evaluation challenging. In the general case, it is impossible to know if a solution exists. Due to the complexity of rule-world pipelines and rulesets, manually fitting a ruleset is unfeasible and algorithmically checking all solutions is intractable. To give some perspective, the OF-DPA TTP is over 12,000 lines of JSON, and real-world rulesets have thousands of rules.

Real-world pipelines can also use non-standard OpenFlow extensions. As an example, OF-DPA provides the VRF field as specialized metadata to carry routing information. Also, the learning table (shown in Figure 2) shares the same lookup table and mirrors the rules in the bridging table. Our implementation does not utilize non-standard extensions.

Fixed-function pipelines usually require redundant operations. As an example, internally, the OF-DPA pipeline requires all packets to include a VLAN header. A rule-fitting solver needs to know to assign a VLAN to all untagged packets and then later pop it, even

SAT Constraints (Cumulative)	Timing (ms) Stage 1 Total	Stage 2 Build SAT	Stage 2 Solve SAT	Stage 2 Verify Sln.	Solver Total	Number of Solutions Candidate	Valid	Unique
One Transformation	32 ±3%	6.7 ±4%	2,325 ±2%	10,960 ±1%	13,323 ±1%	10,000[a]	8,229	7
Placements	32 ±5%	8.3 ±4%	60 ±16%	338 ±3%	439 ±3%	240	155	10
Placement Conflicts	31 ±2%	8.3 ±5%	19 ±12%	142 ±2%	200 ±2%	64	49	10
Table Miss	32 ±5%	8.7 ±5%	15 ±10%	131 ±3%	187 ±3%	49	49	10
Hit Placements	32 ±3%	10 ±16%	4.0 ±10%	88 ±7%	134 ±6%	10	10	10
Forwarding Conflicts	32 ±3%	9.0 ±4%	3.9 ±9%	84 ±3%	128 ±3%	10	10	10

[a]The experiment was limited to the first 10,000 candidate solutions out of 4 million

Table 1: The performance of the rule-fitting solver converting from the 5-table ruleset to the 2-table pipeline when cumulatively adding SAT constraints. Each row shows a breakdown of the solver's run-time and the total number of candidate solutions, valid solutions, and, of those valid solutions, how many were unique. Reading downwards, as we added constraints, the time to build the SAT problem increased slightly; however, the number of candidate solutions and, therefore, the time spent verifying candidates decreased. Overall, the reduced time spent verifying solutions dominated the increased time to build the SAT expression resulting in better performance.

SAT Constraints (Cumulative)	Timing (ms) Stage 1 Total	Stage 2 Build SAT	Stage 2 Solve SAT	Stage 2 Verify Sln.	Solver Total	Number of Solutions Candidate	Valid	Unique
One Transformation	63 ±3%	5.6 ±5%	4.9 ±14%	188 ±3%	262 ±2%	16	1	1
Placements	60 ±0%	6.5 ±3%	6.2 ±6%	176 ±1%	248 ±1%	16	1	1
Placement Conflicts	60 ±1%	6.7 ±4%	2.1 ±6%	62 ±1%	130 ±1%	4	1	1
Table Miss	60 ±1%	7.2 ±4%	2.1 ±7%	62 ±1%	131 ±1%	4	1	1
Hit Placements	60 ±1%	7.8 ±3%	2.1 ±7%	62 ±2%	132 ±1%	4	1	1
Forwarding Conflicts	61 ±1%	7.8 ±2%	3.1 ±7%	63 ±1%	135 ±1%	4	1	1

Table 2: The performance of the rule-fitting solver converting from the 2-table ruleset to the 5-table pipeline comparing different SAT constraints. After adding placement variables, all additional constraints failed to reduce the number of candidate solutions. After which, the run-time increased slightly due to the overhead required to add SAT constraints.

when the original rule does not reference VLANs. As an example, when fitting the Faucet switch ruleset described in §6.2.1 into the OF-DPA pipeline, our solver completes after 5-10 minutes without finding a solution. One cause of the failure is that our solver cannot fit a policy rule applied to both tagged and untagged packets into the OF-DPA ACL table because rules in the ACL table cannot match untagged packets and our solver is not aware all packets entering the ACL table have a VLAN. This problem is solvable by adding the appropriate transformations like those we described in §4.2.6 for the case that the pipeline uses the action-set and requires we add placeholder actions and clear-actions.

Our method is not suitable for flexible pipelines. By design, to explore all possible solutions, our method finds all possible placements of an input rule. Exploring all possibilities works well with constrained pipelines because the number of choices is low, as is often the case with fixed-function pipelines. However, for less constrained pipelines, the number of placements grows too large and exhausts system memory. Further research is required to find the best balance between searching all possible solutions and keeping the problem size tractable when targeting less constrained pipelines.

7 RELATED WORK

Most similar to our work, FlowConvertor [19] (a successor to FlowAdapter [18]) presented an online rule-fitting approach, which used incremental algorithms to maintain the ruleset as a directed acyclic graph. The authors found that the overhead of FlowConvertor was in the order of 1ms when fitting synthetic rulesets to a real-world 2 table pipeline and two synthetic pipelines. While our implementation is not fast enough for online deployment, compared to FlowConvertor, our approach does not require metadata, considers where the target pipeline supports an action, and verifies the equivalence of the solution.

NOSIX [28] and TTPs [9] suggested an alternative architecture to solve device interoperability, where both application developer and vendor agree on a common virtual pipeline yet do not provide algorithms to convert to this common pipeline. TableVisor 2.0 [11] presented an architecture to expose multiple switches, each with limited capabilities, as one more capable switch.

Other research has explored how to split forwarding between multi-table pipelines, without considering device limitations. Sun et al. [24] looked at fitting multiple virtual network tenants into a multi-table pipeline with full OpenFlow support. Similarly, high-level language compilers [21, 27, 28] target multi-table OpenFlow pipelines independent of any particular switch. Our research complements these approaches with the device conversion layer.

Recent research has looked at formalizing aspects of the rule-fitting problem and multi-table pipelines. Leet et al. [14] defined a common space to represent the capabilities of SDN programs and hardware pipelines together in the same domain, to check if hardware can realize an SDN program. Németh et al. [17] applied relational database and formal language theory to transform the table structure of match-action pipelines into more compact forms.

8 CONCLUSION

This paper has presented a novel method to fit existing rulesets into fixed-function pipelines without relying on metadata. This method preprocessed the ruleset and next found the possible transformations of each input rule in isolation. From these transformations, the method used a SAT problem to search for candidate solutions and verified these solutions equivalence against the input ruleset.

We implemented this method in our rule-fitting solver and have released the source code [1] and evaluation artifacts [2]. We evaluated our solver with a synthetic ruleset and pipeline based on OF-DPA, which showed this approach is a viable way to address the complexities of fixed-function pipelines. Towards reducing the size of the problem, we found that our method of compressing the ruleset significantly improved the performance of our solver. In one experiment, with a synthetic ruleset with 50 hosts learned, compression reduced the time to solve from 12min 38s to 6.1s. Additionally, we demonstrate that ruleset compression applies to real-world rulesets and compressed two rulesets by 97.6% and 82.9%. We evaluated the effectiveness of our SAT expression at reducing the problem size and found our SAT constraints excluded only invalid solutions and generally improved the performance of the solver.

Finally, we discussed the difficulties remaining when fitting real-world rulesets and pipelines. In particular, how there is no general way to know if a solution exists, which means if a rule-fitting solver cannot find a solution, it is in general impossible to verify that result.

REFERENCES

[1] [n.d.]. ofsolver. https://github.com/wandsdn/ofsolver

[2] [n.d.]. ofsolver-evaluation. https://github.com/wandsdn/ofsolver-evaluation

[3] Josh Bailey and Stephen Stuart. 2016. Faucet: Deploying SDN in the Enterprise. *Commun. ACM* 60, 1 (Dec 2016), 45–49.

[4] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2018. Control Plane Compression. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication - SIGCOMM '18*. ACM Press, 476–489.

[5] Broadcom. [n.d.]. *Broadcom-Switch/of-dpa.* https://github.com/Broadcom-Switch/of-dpa

[6] Broadcom. 2016. OpenFlow Data Plane Abstraction (OF-DPA) 2.01 Abstract Switch Specification. (Jan 2016). https://github.com/Broadcom-Switch/of-dpa/blob/master/OFDPAS-ETP100-R.pdf

[7] DIMACS Challenge. 1993. Satisfiability: Suggested Format. (May 1993).

[8] Niklas Een, Alan Mishchenko, and Niklas Sörensson. 2007. *Applying Logic Synthesis for Speeding Up SAT.* Vol. 4501. Springer Berlin Heidelberg, 272–286.

[9] Open Networking Foundation. 2014. OpenFlow Table Type Patterns. (Aug 2014). https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/OpenFlowTableTypePatternsv1.0.pdf

[10] Open Networking Foundation. 2015. OpenFlow Switch Specification - Version 1.3.5. (Mar 2015). https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.3.5.pdf

[11] Stefan Geissler, Stefan Herrnleben, Robert Bauer, Steffen Gebert, Thomas Zinner, and Michael Jarschel. 2017. Tablevisor 2.0: Towards full-featured, scalable and hardware-independent Multi Table Processing. In *2017 IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 1–8.

[12] Matt Hamblen. 2019. *Broadcom's new Trident 4 switch stays competitive with Barefoot and Intel.* https://www.fierceelectronics.com/electronics/broadcom-s-new-trident-4-switch-stays-competitive-barefoot-and-intel

[13] Greg Hankins. 2007. FIB Scaling: A Switch/Router Vendor Perspective. https://archive.nanog.org/meetings/nanog39/presentations/fib-hankins.pdf

[14] Christopher Leet, Xin Wang, Y. Richard Yang, and James Aspnes. 2018. Toward the First SDN Programming Capacity Theorem on Realizing High-Level Programs on Low-Level Datapaths. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*. IEEE, 711–719.

[15] Fábián Tamás László. [n.d.]. *SATisPy.* https://github.com/netom/satispy

[16] Big Switch Networks. 2019. *Hardware Compatibility List.* https://opennetlinux.org/hcl.html

[17] Felicián Németh, Marco Chiesa, and Gábor Rétvári. 2019. Normal Forms for Match-Action Programs. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies (CoNEXT '19)*. ACM, 44–50.

[18] 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 *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking - HotSDN '13*. ACM Press, 85–90.

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

[20] Pica8. 2016. *SDN System Performance | Pica8.* https://web.archive.org/web/20160801084903/http://www.pica8.com/pica8-deep-dive/sdn-system-performance

[21] Joshua Reich, C. Monsanto, Nate Foster, Jennifer Rexford, and D. Walker. 2013. Modular SDN Programming with Pyretic. *USENIX Login* 38 (2013), 128–134.

[22] Richard Sanger, Matthew Luckie, and Richard Nelson. 2019. Identifying Equivalent SDN Forwarding Behaviour. In *Proceedings of the 2019 ACM Symposium on SDN Research - SOSR '19*. ACM Press, 127–139.

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

[24] Xiaoye Sun, T.S. Eugene Ng, and Guohui Wang. 2015. Software-Defined Flow Table Pipeline. In *2015 IEEE International Conference on Cloud Engineering*. IEEE, 335–340.

[25] Faraz Taifehesmatian. 2019. Cisco Nexus 3000 Switch Architecture. https://www.ciscolive.com/c/dam/r/ciscolive/us/docs/2018/pdf/BRKDCN-3734.pdf

[26] Thomas Vachuska and Pradeep Reddy Kandi. 2016. *Device Driver Subsystem - ONOS - Wiki.* https://wiki.onosproject.org/display/ONOS/Device+Driver+Subsystem

[27] Junchang Wang, Shaojin Cheng, and Xiong Fu. 2018. SDN Programming for Heterogeneous Switches with Flow Table Pipelining. *Scientific Programming* 2018 (Nov 2018), 1–13.

[28] Minlan Yu, Andreas Wundsam, and Muruganantham Raju. 2014. NOSIX: A Lightweight Portability Layer for the SDN OS. *ACM SIGCOMM Computer Communication Review* 44, 2 (Apr 2014), 28–35.

A BOOLEAN AND SET NOTATION

Boolean Operators

Name	Symbol
Negation	$\neg$
And (Conjunction)	$\wedge$
Or (Disjunction)	$\vee$
Exclusive or (Xor)	$\oplus$
Implies	$\rightarrow$
Equivalence	$\leftrightarrow$

Set Operations

Name	Symbol
Union	$\cup$
Intersection	$\cap$
Element In	$\in$
Subset	$\subseteq$
For All	$\forall$

Boolean Set Flattening

Name	Symbol	Description
Big And	$\bigwedge\{...\}$	$\bigwedge\{a, b, c\}$ is equivalent to $a \wedge b \wedge c$
Big Or	$\bigvee\{...\}$	$\bigvee\{a, b, c\}$ is equivalent to $a \vee b \vee c$
One Hot	$\textsc{onehot}(\{...\})$	Satisfied if only one item is true