Реализация нескольких виртуальных сетей в программно-конфигурируемой сети

Аннотация. Software Defined Networks (SDN) является одной из современных технологий виртуализации сети. При реализации виртуальной сети на плоскости данных SDN могут возникать нежелательные эффекты: появление непредусмотренных путей, по которым пересылаются пакеты, «зацикливание», когда пакеты бесконечно перемещаются по циклическому пути и бесконечно клонируются в точке ответвления от цикла конечного отрезка пути до хоста, дублирующие пути, когда хост получает один и тот же пакет несколько раз. Мы также показываем, что эти эффекты могут возникать при совместной реализации нескольких виртуальных сетей даже в том случае, когда реализация каждой отдельной виртуальной сети не имеет этих эффектов. Предлагаются методы верификации реализации как отдельной виртуальной сети, так и совместной реализации нескольких виртуальных сетей. Также устанавливается достаточное условие на граф физических связей плоскости данных SDN, при выполнении которого любой набор виртуальных сетей может быть реализован без возникновения указанных эффектов.

Ключевые слова: Software Defined Networks (SDN), Network Virtualization, Graph paths, Edge Simple Paths, Arc Closure, Verification;

1. **Введение**

Программно-конфигурируемые сети с разделенными плоскостями данных и управления являются одной из основных технологий реализации виртуальных сетей. На плоскости данных хосты обмениваются между собой пакетами, которые проходят те или иные пути через промежуточные коммутаторы. Коммутатор, принимая пакет, пересылает его без изменений одному или нескольким своим соседям (хостам или коммутаторам) в зависимости от того, от какого соседа (хоста или коммутатора) он принял пакет, и от значений параметров в заголовке пакета. Если коммутатор пересылает пакет нескольким своим соседям, то это означает, что пакет клонируется и далее его клоны перемещаются по сети независимо друг от друга. Правила работы коммутатора устанавливаются контроллером при настройке коммутатора. Совокупность значений параметров в заголовке пакета, влияющих на пересылку пакетов, называют идентификатором пакета. Пакеты с разными идентификаторами перемещаются по сети независимо друг от друга и проходят, вообще говоря, разные пути от хостов-отправителей к хостам-получателям.

Виртуальная сеть описывается множеством (упорядоченных) пар хостов (отправитель, получатель) и указанием пакетов, которые могут передаваться от отправителя к получателю, и реализуется указанием множества путей, по которым могут двигаться пакеты с теми или иными идентификаторами. Это, в свою очередь, определяет те правила, которые должны быть установлены в коммутаторах при их настройке.

При реализации виртуальной сети возникает проблема непредусмотренных путей. Это, во-первых, дублирующие пути, когда хост получает один и тот же пакет несколько раз. Во-вторых, что более важно, циклов, по которым пакеты бесконечно перемещаются и бесконечно клонируются в точке разветвления цикла и конечного отрезка пути к хосту-получателю, что приводит к бесконечному числу дублирующих путей. Эта проблема исследовалась в ряде работ [1, 2, 3, 4, 5, 6, 7], в которых были предложены различные её решения.

Однако проблема непредусмотренных путей возникает ещё раз при реализации нескольких виртуальных сетей. Даже в том случае, когда реализация каждой из этих виртуальных сетей не порождает непредусмотренных путей, их совместная реализация такие пути может порождать. Это объясняется взаимным влиянием реализаций нескольких виртуальных сетей, если какие-то идентификаторы пакетов используются в реализациях нескольких виртуальных сетей. Данная статья посвящена исследованию этого влияния. Основное внимание уделяется проблеме появления циклов, составленных из отрезков путей, появляющихся в реализациях разных виртуальных сетей для одних и тех же идентификаторов пакетов.

Структура статьи следующая. После введения в разделе 2 даются необходимые формальные определения и формальная постановка задачи. Разделы 3–5 посвящены верификации заданных реализаций виртуальных сетей и, прежде всего, проверке отсутствия циклов. В разделе 3 описывается алгоритм проверки наличия непредусмотренных путей и, в частности, циклов при реализации одной виртуальной сети, предложенный в [3, 6]. В разделе 4 предлагается алгоритм поиска всех циклов, порождаемых реализацией нескольких виртуальных сетей с определением идентификаторов тех пакетов, которые могут двигаться по этим циклам. Раздел 5 содержит более эффективный алгоритм определения наличия или отсутствия циклов, но при наличии циклов определяются не обязательно все циклы и не все идентификаторы пакетов, двигающих по циклам. В разделе 6 определяется достаточное условие возможности реализации любого набора виртуальных сетей без порождения непредусмотренных путей. Это условие является расширением аналогичного условия для реализации одной виртуальной сети, предложенного в [4, 5, 7]. В заключении подводятся итоги проделанной работы и определяются направлениях дальнейших исследований.

1. **Формальные определения и постановка задачи**

Плоскость данных программно-конфигурируемой сети моделируется конечным неориентированным связным графом *G* = (*V*, *E*) без кратных ребер и петель, где *V* множество вершин, которыми являются хосты и коммутаторы, а *E* ⊆ *V* × *V* множество ребер, отображающих физические каналы связи между вершинами. Мы предполагаем, что каждый хост соединен только с одним коммутатором. Без ограничения общности можно считать, что все вершины степени 1 являются хостами.

Поскольку ребра неориентированные и нет кратных ребер, ребро можно задавать (неупорядоченной) парой вершин *a* и *b*, которые оно соединяет: *ab* или *ba*. Путь однозначно задаётся последовательностью вершин, через которые он проходит. Путь из вершины *a* в вершину *b* называют *ab*-путем. Путь называется *полным*, если он начинается и заканчивается в хосте, а все промежуточные вершины — коммутаторы. Путь, в котором все вершины (дуги) попарно разные, называется *вершинно-простым* (*реберно-простым*).

Вершины графа будем обозначать строчными буквами *a*, *b*, … *y*, *z*, пути – жирными строчными буквами ***p***, ***q***, ***r***,…, множества – прописными буквами – *A*, *B*, … *Y*, *Z*, семейства множеств – прописными буквами курсивом – *A*, *B*, … *Y*, *Z*.

Правило коммутатора *s* имеет вид ι:*asb*, где ι идентификатор пакета, *a* и *b* вершины (коммутаторы или хосты), соединенные ребрами с коммутатором *s*. Такое правило означает, что коммутатор *s*, получив пакет с идентификатором ι от своего соседа *a*, пересылает его без изменений соседу *b*. Клонирование пакета происходит тогда, когда в коммутаторе *s* имеется несколько правил, отличающихся только соседом *b*.

**Proposition 1.** Полный путь *a*1...*an* реализуется для пакетов с идентификатором ι тогда и только тогда, когда в каждом коммутаторе *ai*, *i* = 2..*n*–1, есть правило ι:*a*i–1*aiai*+1.

**Corollary 1.** Если для некоторого идентификатора ι реализуются два полных пути ***p****ab****q*** и ***p****'ab****q****'*, проходящие через общую дугу *ab*, то для этого идентификатора ι реализуются и два полных пути ***p****ab****q****'* и ***p****'ab****q***.

Тем самым при реализации множества *P* полных путей порождается соответствующее множество *P*↓ правил коммутаторов, которое в свою очередь реализует надмножество множества *P*, называемое *замыканием по дугам* множества *P* и обозначаемое *P*↓↑.

**Proposition 2.** Множество *P*↓↑ порождается следующими правилами вывода:

***p*** ∈ *P* влечет ***p*** ∈ *P*↓↑,

***p****ab****q*** ∈ *P*↓↑ & ***p`****ab****q`*** ∈ ***P***↓↑  влечет ***p****ab****q`*** ∈ *P*↓↑.

Реализация виртуальной сети задаётся парой множеств (*Z*, *P*), где *Z* множество идентификаторов пакетов, *P* множество полных путей. Пакет с идентификатором из множества *Z* проходит по пути из множества *P*↓↑, если этот пакет посылает хост в начале этого пути.

**Proposition 3.** Множество *P* не порождает непредусмотренных путей тогда и только тогда, когда *P* = *P*↓↑, т.е. множество *P* замкнуто по дугам.

Само по себе порождение непредусмотренных путей может не считаться проблемой, но при обязательном условии, что не возникают циклы. Цикл возникает, если не все реализуемые пути являются реберно-простыми. Не реберно-простой путь ***p****ab****q****ab****r*** при замыкании по дугам порождает бесконечное число путей вида ***p****ab*(***q****ab*)*n****r***, где *n* число проходов по циклу ***q****ab*.

**Proposition 4.** Множество *P*↓↑ конечно тогда и только тогда, когда все его пути реберно-простые.

В следующем разделе описывается алгоритм проверки наличия непредусмотренных путей и циклов для заданной пары (*Z*, *P*).

Если задаётся реализация *n* виртуальных сетей, то это значит, что задаются два равномощных семейства множеств Z = { *Z*1, ..., *Zn* } и P = { *P*1, ..., *Pn* }; где реализация *i*-й виртуальной сети задана парой (*Zi*, *Pi*), *i* = 1..*n*. Ставится задача определить, возникают ли циклы при совместной реализации этих виртуальных сетей. Для этого предлагаются два алгоритма в разделах 4 и 5.

1. **Верификация одной виртуальной сети**

Пусть задан граф физических связей *G* = (*V*, *E*) и реализация виртуальной сети (*Z*, *P*). Требуется проверить: 1) порождаются ли непредусмотренные пути, 2) порождаются ли циклы, т.е. не реберно-простые пути. Непредусмотренные пути возникают, если *P* ≠ *P*↓↑. Поскольку *P* ⊆ *P*↓↑, неравенство *P* ≠ *P*↓↑ эквивалентно неравенству |*P*| ≠ |*P*↓↑|. Циклы возникают, если *P*↓↑ бесконечно.

В [3, 6] предложен алгоритм верификации, основанный на ориентированном графе *L*(*P*), который представляет собой подграф реберного графа для подграфа графа *G*, порожденного множеством путей *P*. Вершинами графа *L*(*P*) являются дуги путей из *P*, а также две дополнительные вершины *source* и *sink*. Дуга (*ab*, *b'c*) в графе *L*(*P*) соответствует пути длиной два в графе *G*, т.е. *b* = *b'*, и проводится тогда и только тогда, когда в *P* есть путь, проходящий отрезок *abc*. Дуги вида (*source*, *xa*) ведут из вершины *source* во все вершины *xa*, где *x* начальный хост пути из *P*. Дуги вида (*ax*, *sink*) ведут из всех вершин *ax*, где *x* конечный хост пути из *P*, в вершину *sink*.

**Proposition 5.** Непредусмотренные пути могут порождаться только путями длины больше 2.

Если полный путь в *P*↓↑ непредусмотренный (отсутствует в *P*), то он имеет вид ***p****ab****q****'* или ***p****'ab****q*** и порождается двумя полными путями ***p****ab****q*** и ***p****'ab****q****'*, причём ***p*** ≠ ***p****'* и ***q*** ≠ ***q****'*. Поэтому ***p*** или ***p****'*, а также ***q*** или ***q****'* имеет ненулевую длину. Отсюда, поскольку пути ***p****ab****q*** и ***p****'ab****q****'* полные, то *a* и *b* коммутаторы, а тогда все четыре отрезка ***p***, ***p****'*, ***q***, ***q****'* имеют ненулевую длину и, следовательно, пути ***p****ab****q*** и ***p****'ab****q****'* имеют длину больше 2. Отсюда следует, что достаточно ограничиться путями из *P*, имеющими длину больше 2.

**Algorithm 1:** Derivation a graph *L*(*P*) = (*VL*, *EL*)

**Input:** A set *P* of complete paths

**Output:** A graph *L*(*P*)

Derive a subset *Q* = {*q*1, ..., *qk*} of *P* that contains all the paths of length greater than two; we denote as *kj* the length of a path *qj*, *j* ∈ {1, ..., *k*};

*VL* = {*source*, *sink*};

*EL* = ∅;

*j* = 1;

***while*** *j* < *k* ***do***

⎢ *VL* = *VL* ∪ {(*qj*(1), *qj*(2))};

⎢ *EL* = *EL* ∪ {(*source*, (*qj*(1), *qj*(2))};

⎢ *m* = 2;

⎢ ***while*** *m* < *kj*+1 ***do***

⎢ ⎢ *VL* = *VL* ∪ {(*qj*(*m*), *qj*(*m*+1))};

⎢ ⎢ *EL* = *EL* ∪ {((*qj*(*m*–1), *qj*(*m*)), (*qj*(*m*), *qj*(*m*+1)))};

⎢ ⎣ *m*++;

⎢ *EL* = *EL* ∪ {(*qj*(*kj*), *qj*(*kj*+1)), *sink*)};

⎣ *j*++;

***return*** *L*(*P*);

**Proposition 6.** Алгоритм 1 возвращает граф *L*(*P*) и имеет сложность ***O***(*L*2), где *L* сумма длин путей из *P*.

Сложность алгоритма 1 определяется числом попарных сравнений на равенство дуг путей из *P*, которое имеет оценку ***O***(*L*2).

Полным путям из *P*↓↑ длиной больше 2 взаимно-однозначно соответствуют пути графа *L*(*P*) от вершины *source* до вершины *sink*. При этом реберно-простым путям из *P*↓↑ соответствуют вершинно-простые пути графа *L*(*P*). Таким образом, для проверки наличия или отсутствия непредусмотренных путей в *P*↓↑ достаточно подсчитать число путей в *L*(*P*) от *source* до *sink* и сравнить его с число путей в *P*. Не реберно-простые пути в *P*↓↑ есть тогда и только тогда, когда в графе *L*(*P*) имеются циклы.

Для верификации графа *L*(*P*) можно использовать DFS-алгоритм обхода графа, модифицированный для обнаружения циклов и подсчёта числа путей от *source* до *sink*. DFS-алгоритм работает следующим образом. Вершины графа *L*(*P*) будем окрашивать в белый, чёрный и серый цвета. Сначала все вершины белые. Выбираем вершину *source*, перекрашиваем ее в чёрный цвет и двигаемся по дугам, перекрашивая их концы в чёрный цвет и запоминая в каждой вершине дугу, по которой в неё пришли, как *обратную* дугу (*back*), до тех пор, пока не попадём в чёрную или серую вершину. Если попали в чёрную вершину, мы обнаружили ориентированный цикл. Если попали в серую вершину, возвращаемся по дуге обратно. Если из текущей вершины никакие дуги не ведут в белые вершины (в частности, из вершины *sink*), перекрашиваем вершину в серый цвет и, если эта вершина не является вершиной *source*, возвращаемся по обратной дуге (в направлении противоположном ее ориентации). Если же текущая вершина — это вершина *source*, то алгоритм заканчивает работу.

Для подсчета числа путей в графе *L*(*P*) от *source* до *sink* при отсутствии циклов связываем с каждой вершиной *v* переменную *s*(*v*), равную числу обнаруженных путей из *v* до *sink*. Вначале *s*(*sink*) = 1, а в любой другой вершине *v* инициализируем *s*(*v*) = 0. Когда из вершины *v* по только что пройденной дуге или по обратной дуге возвращаемся в вершину *w*, корректируем *s*(*w*) = *s*(*w*) + *s*(*v*). В конце работы алгоритма, если циклов не обнаружено, число путей от *source* до *sink* равно *s*(*source*).

**Algorithm 2:** Verification of graph *L*(*P*) = (*VL*, *EL*)

**Input:** A graph *L*(*P*)

**Output:** A verdict 1 whether the set *P* is arc closed and a verdict 2 whether the set *P* is finite

Для вершины *v* ∈ *VL* через *d*(*v*) обозначим число дуг, выходящих из *v*, и через *v*1, ..., *vd*(*v*) обозначим конечные вершины этих дуг

*color*(*source*) = *black*;

*out*(*source*) = 1;

*s*(*source*) = 0;

*current* = *source*;

***while*** *current* ≠ *source* ∨ *out*(*current*) < *d*(*current*) ***do***

⎢ ***if*** *out*(*current*) < *d*(*current*) ***then***

⎢ ⎢ *out*(*current*)++;

⎢ ⎢ *x* = *currentout*(*current*);

⎢ ⎢ ***if*** *color*(*x*) = *white* ***then***

⎢ ⎢ ⎢ *color*(*x*) = *black*;

⎢ ⎢ ⎢ *out*(*x*) = 1;

⎢ ⎢ ⎢ *back*(*x*) = *current*;

⎢ ⎢ ⎢ ***if*** *x* = *sink* ***then***

⎢ ⎢ ⎢ ⎣ *s*(*x*) = 1;

⎢ ⎢ ⎢ ***else*** *s*(*x*) = 0;

⎢ ⎢ ⎣ *current* = *x*;

⎢ ⎢ ***else***

⎢ ⎢ ⎢ ***if*** *color*(*x*) = *black* ***then***

⎢ ⎢ ⎢ ⎣ ***return*** (*False*, *False)*;

⎢ ⎢ ⎢ ***else***

⎢ ⎢ ⎢ ⎢ *s*(*current*) = *s*(*current*) + *s*(*x*);

⎢ ⎣ ⎣ ⎣ *out*(*current*)++;

⎢ ***else***

⎢ ⎢ *color*(*current*) = *grey*;

⎢ ⎢ *x* = *back*(*current*);

⎢ ⎢ *s*(*x*) = *s*(*current*) + *s*(*x*);

⎣ ⎣ *current* = *x*;

***if*** *s*(*source*) = |*P*| ***then***

⎣ ***return*** (*True*, *True)*;

***else***

⎣ ***return*** (*False*, *True)*;

**Proposition 7.** Алгоритм 2 возвращает вердикт 1 True тогда и только тогда, когда множество *P* замкнуто по дугам, и возвращает вердикт 2 True тогда и только тогда, когда *P*↓↑ конечно. Сложность алгоритма 2 равна ***O***(*L*).

Время работы DFS-алгоритма на графе *L*(*P*) равно ***O***(*m*), где *m* – число дуг графа *L*(*P*). Очевидно, что число дуг графа *L*(*P*) не превосходит ***O***(|*V*|3). С другой стороны, *m* = ***O***(*L*2), где *L* сумма длин путей из множества *P*.

**Proposition 8.** Суммарная сложность алгоритмов 1 и 2 равна ***O***(*L*2).

1. **Верификация множества виртуальных сетей. Алгоритм 3**

Пусть задан граф физических связей *G* = (*V*, *E*) и *n* виртуальных сетей как пара равномощных семейств: семейство множеств идентификаторов Z = { *Z*1, ..., *Zn* } и семейство множеств путей P = { *P*1, ..., *Pn* }. Пара (*Zi*, *Pi*), *i* = 1..*n*, задаёт реализацию *i*-й виртуальной сети. Требуется определить, возникают ли непредусмотренные пути и циклы при совместной реализации этих виртуальных сетей и, если возникают, то для каких идентификаторов пакетов.

Если бы семейство Z было разбиением объединения входящих в него множеств, т.е. разбиением множества ∪Z, было бы достаточно верифицировать независимо каждую реализацию виртуальной сети (*Zi*, *Pi*), *i* = 1..*n*, поскольку для *i* ≠ *j* множества *Zi* и *Zj* не имели бы общих идентификаторов. Но в общем случае семейство Z может быть покрытием объединения ∪Z. Поэтому задача сводится к построению по этому покрытию разбиения A = { *A*1, ..., *Am* }, где *m* число элементов разбиения, и независимой верификации элементов этого разбиения с помощью алгоритмов 1 и 2.

Каждому элементу *Aj* разбиения A, *j* = 1..*m*, соответствует множество *Uj* индексов по семейству *Z* так, что *Aj* = (∩{ *Zi* | *i* ∈ *Uj* }) \ (∪{ *Zi* | *i* ∉ *Uj* }) не пусто. Соответствующее элементу *Aj* множество *Rj* путей вычисляется как ∪{ *Pi* | *i* ∈ *Uj* }, что требует не более *n* операций объединения, а суммарно соответствующее семейство путей R = { *R*1, ..., *Rm* } вычисляется за ***O***(*nm*).

Построение разбиения A по покрытию Z можно делать итеративно следующим образом. Пусть для первых *i* элементов покрытия построено разбиение их объединения и оно содержит *xi* элементов. Пусть также построено объединение этих элементов разбиения, т.е. объединение первых *i* элементов покрытия. Добавляем *i*+1‑й элемент покрытия. Для этого нужно построить пересечение *i*+1‑го элемента покрытия с каждым *j*-м элементом разбиения, *j* = 1..*xi*, и, если пересечение не пусто, построить также разность *j*-го элемента разбиения и *i*+1‑го элемента покрытия. Кроме этого, нужно построить разность *i*+1‑го элемента покрытия и объединения первых *i* элементов покрытия, а также объединение первых *i*+1 элементов покрытия как объединение *i*+1‑го элемента покрытия и объединения первых *i* элементов покрытия.

**Algorithm 3:** Derivation a partition from indexed family of sets

**Input:** A indexed families of sets Z = { *Z*1, ..., *Zn* } and P = { *P*1, ..., *Pn* }

**Output:** A partition A = { *A*1, ..., *Am* } and corresponding indexed family of sets R = { *R*1, ..., *Rm* }

U = (*U*1, …, *U*|U |), где *Uj* ⊆ {1..*n*} для *j* = 1..|U|, — набор подмножеств индексов по *Z*, соответствующий текущему (построенному) разбиению A = (*A*1, …, *A*|U |), где *Aj* = (∩{ *Zi* | *i* ∈ *Uj* }) \ (∪{ *Zi* | *i* ∉ *Uj* }) для *j* = 1..|U|;

*B* = ∪A;

U ′ и A ′ — U и A, соответствующие строящемуся разбиению;

*U* = (); A = (); *B* = ∅; U ′ = (); A ′ = ();

***for*** *i* = 1..*n* ***do***

⎢ ***for*** *j* = 1..|U| ***do***

⎢ ⎢ *X* = *Aj* ∩ *Zi*;

⎢ ⎢ ***if*** *X* = ∅ ***then***

⎢ ⎢ ⎢ A ′ = A ′⋅(*Aj*);

⎢ ⎢ ⎣U ′ = U ′⋅(*Uj*);

⎢ ⎢ ***else*** A ′ = A ′⋅(*X*);

⎢ ⎢ ⎢ U ′ = U ′⋅(*Uj* ∪ {*i*});

⎢ ⎢ ⎢ *X* = *Aj* \ *Zi*;

⎢ ⎢ ⎢ ***if*** *X* ≠ ∅ ***then***

⎢ ⎢ ⎢ ⎢ A ′ = A ′⋅(*X*);

⎢ ⎣ ⎣ ⎣ U ′ = U ′⋅(*Uj*);

⎢ *X* = *Zi* \ *B*;

⎢ ***if*** *X* ≠ ∅ ***then***

⎢ ⎢ A ′ = A ′⋅(*X*);

⎢ ⎢ U ′ = U ′⋅({*i*});

⎢ ⎣ *B* = *B* ∪ *X*;

⎢ A = A ′; A ′ = ();

⎣ U = U ′; U ′ = ();

R = { ∪{ *Pi* | *i* ∈ *Uj* } | *j* = 1..|U | };

***return*** (A, R);

**Proposition 9.** Алгоритм 3 возвращает разбиение A и соответствующее ему семейство путей R. Сложность алгоритма равна ***O***(*nm*) = ***O***(*nk*), где *m* число элементов разбиения, *k* = |∪Z| число различных идентификаторов в семействе Z.

На *i*+1-м шаге мы выполняем не более 2*xi* + 2 операций (пересечение, разность или объединение двух множеств), и к разбиению добавляем число элементов, которое может колебаться от 0 до *xi* + 1. Тем самым, *xi* ≤ *xi*+1 ≤ 2*xi* + 1, *i* = 1..*n*–1. Обозначим через *yi* суммарное число операций к концу *i*-го шага. Имеем: *x*1 = 1, *y*1 = 0 — на первом шаге просто выбираем первый элемент покрытия; *xi* ≤ *xi*+1 ≤ 2*xi* + 1 для *i* = 1..*n*‑1; *m* = *xn*, *yn* ≤ (2*x*1 + 2) + … + (2*xn*‑1 + 2) = 2(*x*1 + … + *xn*‑1) + 2(*n*‑1) ≤ 2(*xn* + … + *xn*) + 2(*n*‑1) = 2(*n*‑1)*xn* + 2(*n*‑1) = 2(*n*‑1)(*xn* + 1) = 2(*n*‑1)(*m* + 1). Эта оценка для данного *n* достигается, когда все *xi* равны 1: покрытие состоит из *n* одинаковых множеств, *m* = 1, *yn* = 4*n* – 2.

Тем самым, сложность построения разбиения равна *yn* = ***O***(*nm*). Заметим, что число *m* элементов разбиения не превосходит числа *k* = |∪Z| различных идентификаторов в семействе Z. Поэтому *yn* = ***O***(*nk*).

**Proposition 10.** Суммарная сложность построения разбиения и верификации его элементов равна ***O***(*mL*2) = ***O***(*kL*2).

Построение разбиения A и соответствующего семейства путей R имеет сложность ***O***(*nm*), построение графа *L*(*P*) и верификация по этому графу для одного элемента разбиения имеет сложность ***O***(*L*2), а число таких элементов равно *m*. Тем самым, суммарная сложность построения разбиения и верификации его элементов равна ***O***(*nm* + *mL*2). Без ограничения общности можно считать, что все множества семейства P не пусты, а тогда *n* ≤ *L* и ***O***(*nm* + *mL*2) = ***O***(*mL*2) = ***O***(*kL*2).

1. **Верификация множества виртуальных сетей. Алгоритм 4**

Пусть задан граф физических связей *G* = (*V*, *E*) и *n* виртуальных сетей как пара равномощных семейств: семейство множеств идентификаторов Z = { *Z*1, ..., *Zn* } и семейство множеств путей P = { *P*1, ..., *Pn* }. Пара (*Zi*, *Pi*), *i* = 1..*n*, задаёт реализацию *i*-й виртуальной сети. Требуется определить, возникают ли циклы при реализации этих виртуальных сетей. В отличие от постановки задачи в предыдущем разделе не требуется определять наличие или отсутствие непредусмотренных реберно-простых путей, и не требуется определять идентификаторы пакетов, которые могут двигаться по циклам.

Идея предлагаемого ниже алгоритма 4 заключается в следующем. Пусть есть два вложенных одно в другое подсемейств V ⊆ W ⊆ Z такие, что пересечения входящих в них множеств не пусты: ∩V ≠ ∅ (и, следовательно, ∩W ≠ ∅). Эти подсемейства можно представить в виде V = { *Zi* | *i* ∈ *U*V } и W = { *Zi* | *i* ∈ *U*W }, где *U*V, *U*W ⊆ {1..*n*} и *U*V ⊆ *U*W. Тогда соответствующие множества путей тоже вложены одно в другое: *R*V ⊆ *R*W, где *R*V = { *Pi* | *i* ∈ *U*V } и *R*W = { *Pi* | *i* ∈ *U*W }. Если не реберно-простой путь есть в *R*V↓↑, то он есть и в *R*W↓↑. Поэтому достаточно искать циклы в множествах путей, соответствующих максимальным по вложенности подсемействам семейства Z с непустым пересечением множеств, входящих в каждое подсемейство.

Для сравнения сложности верификации по алгоритмам 3 и 4 важным является число строящихся множеств путей, для каждого из которых нужно строить граф *L*(*P*) и его верифицировать. Для алгоритма 3 это число элементов разбиения, которое имеет максимальное значение 2*n*–1. Для алгоритма 4 это число максимальных по вложенности элементов разбиения. Поскольку такие элементы образуют антицепь по отношению вложенности, их число не превосходит длину максимальной антицепи. Это также называют шириной булевой решетки *Bn*, которая, по теореме Спернера (Sperner's theorem [8]), не превосходит *Cn*⎣*n*/2⎦. Имеем C*nn*/2 ~ 4*n*/2/(π(*n*/2))1/2 = (1/((π/2)1/2))2*n*. Это в (π/2)1/2 ≈ 1,25\**n*1/2 раз меньше, чем 2*n*–1 для алгоритма 3.

**Proposition 11.** Пересечение множеств максимального по вложенности подсемейства семейства Z с непустым пересечением составляющих подсемейство множеств является элементом разбиения A.

Действительно, пусть есть некоторое максимальное по вложенности подсемейство V ⊆ Z с непустым пересечением ∩V ≠ ∅. Тогда в силу максимальности по вложенности подсемейства V для любого *X* ∈ Z \ V имеет место (∩V) ∩ *Х* = ∅. Отсюда (∩V) ∩ (∪(Z \ V)) = ∅. Следовательно, (∩V) \ (∪(Z \ V)) = ∩V ≠ ∅. А это значит, что ∩V есть элемент разбиения A.

В алгоритме 3 строится семейство U = (*U*1, …, *Um*), где *Uj* ⊆ {1..*n*} для *j* = 1..*m*, — набор подмножеств индексов по *Z*, соответствующий разбиению A = (*A*1, …, *Am*), где *Aj* = (∩{ *Zi* | *i* ∈ *Uj* }) \ (∪{ *Zi* | *i* ∉ *Uj* }) ≠ ∅ для *j* = 1..*m*, и соответствующее семейство путей R = (*R*1, …, *Rm*), где *Rj* = { ∪{ *Pi* | *i* ∈ *Uj* }. В силу доказанного достаточно искать циклы в множествах путей *Rj*, соответствующих максимальным по вложенности множествам *Uj*.

**Algorithm 4:** Derivation a indexed family R of sets of paths, соответствующее максимальным по вложенности подсемействам семейства Z с непустым пересечением множеств, входящих в каждое подсемейство.

**Input:** A indexed families of sets Z = { *Z*1, ..., *Zn* } and P = { *P*1, ..., *Pn* }

**Output:** A indexed family R = { *R*1, ..., *Rm'* }

U = (*U*1, …, *U*|U |), где *Uj* ⊆ {1..*n*} для *j* = 1..|U|, — набор подмножеств индексов по *Z*, соответствующий текущему (построенному) разбиению A = (*A*1, …, *A*|U |), где *Aj* = (∩{ *Zi* | *i* ∈ *Uj* }) \ (∪{ *Zi* | *i* ∉ *Uj* }) для *j* = 1..|U|;

*B* = ∪A;

U ′ и A ′ — U и A, соответствующие строящемуся разбиению;

U = (); A = (); *B* = ∅; U ′ = (); A ′ = ();

***for*** *i* = 1..*n* ***do***

⎢ ***for*** *j* = 1..|A| ***do***

⎢ ⎢ *X* = *Aj* ∩ *Zi*;

⎢ ⎢ ***if*** *X* = ∅ ***then***

⎢ ⎢ ⎢ A ′ = A ′⋅(*Aj*);

⎢ ⎢ ⎣ U′ = U′⋅(*Uj*);

⎢ ⎢ ***else*** A ′ = A ′⋅(*X*);

⎢ ⎢ ⎢ U′ = U′⋅(*Uj* ∪ {*i*});

⎢ ⎢ ⎢ *X* = *Aj* \ *Zi*;

⎢ ⎢ ⎢ ***if*** *X* ≠ ∅ ***then***

⎢ ⎢ ⎢ ⎢ A ′ = A ′⋅(*X*);

⎢ ⎣ ⎣ ⎣ U′ = U′⋅(*Uj*);

⎢ *X* = *Zi* \ *B*;

⎢ ***if*** *X* ≠ ∅ ***then***

⎢ ⎢ A ′ = A ′⋅(*X*);

⎢ ⎢ U′ = U′⋅({*i*});

⎢ ⎣ *B* = *B* ∪ *X*;

⎢ *A* = A ′; A ′ = ();

⎣ U = U′; U′ = ();

*i* = 1;

***while*** *i* < |U| ***do***

⎢ *j* = *i*+1;

⎢ ***while*** *j* ≤ |U| ***do***

⎢ ⎢ ***if*** *Ui* ⊆ *Uj* ***then***

⎢ ⎢ ⎢ U = U \ {*Ui*};

⎢ ⎢ ⎣ ***break***;

⎢ ⎢ ***else*** ***if*** *Uj* ⊂ *Ui* ***then***

⎢ ⎢ ⎢ ⎣ U = U \ {*Uj*};

⎢ ⎣ ⎣ ***else*** *j* = *j*+1;

⎢ ***if*** *j* > |U| ***then***

⎣ ⎣ *i* = *i*+1;

R = { ∪{ *Pi* | *i* ∈ *Uj* } | *j* ∈ 1..|U| };

***return*** R;

**Proposition 12.** Алгоритм 4 возвращает семейство путей R. , соответствующее максимальным по вложенности подсемействам семейства Z с непустым пересечением множеств, входящих в каждое подсемейство. Сложность алгоритма равна ***O***(*nm* + *m*2) = ***O***(*nk* + *k*2), где *m* число элементов разбиения, *k* = |∪Z| число различных идентификаторов в семействе Z.

Разбиение строится за время ***O***(*nm*). Далее попарно сравнивая множества семейства U, за время ***O***(*m*2) выбираем подсемейство из *m'* максимальных по вложенности множеств индексов, по которому за время ***O***(*nm'*) строим искомое семейство R множеств путей. Учитывая, что *m'* ≤ *m* ≤ *k*, суммарная сложность равна ***O***(*nm* + *m*2 + *nm*) = ***O***(*nm* + *m*2) = ***O***(*nk* + *k*2).

**Proposition 13.** Суммарная сложность алгоритма 4 и последующей верификации равна ***O***(*mL*2) = ***O***(*kL*2).

Сложность алгоритма 4 равна ***O***(*nm* + *m*2), построение графа *L*(*P*) и верификация по этому графу для одного множества путей имеет сложность ***O***(*L*2), а число таких элементов равно *m'* ≤ *m* ≤ *k*. Тем самым, суммарная сложность построения и верификации равна ***O***(*nm* + *m*2 + *mL*2). Без ограничения общности можно считать, что все множества семейства P не пусты, а тогда *n* ≤ *L*, *m* ≤ *L* и ***O***(*nm* + *m*2 + *mL*2) = ***O***(*mL*2) = ***O***(*kL*2).

1. **Достаточное условие возможности реализации любого набора виртуальных сетей без непредусмотренных путей**

В [4, 5] исследуется следующая проблема: какими свойствами должен обладать граф *G* физических связей, чтобы была возможна реализация любой виртуальной сети без порождения непредусмотренных путей, дублирующих путей и циклов.

Виртуальная сеть описывается парой (*Z*, *D*), где *Z* множество идентификаторов пакетов, которые могут пересылаться по этой виртуальной сети, а *D* множество (упорядоченных) пар хостов (отправитель, получатель). Множество *D* пар хостов называется *нормальным*, если оно не содержит пары одинаковых хостов (*x*, *x*). Реализация виртуальной сети (*Z*, *D*) — это пара (*Z*, *P*), где *P* множество полных путей, связывающих все пары хостов из *D*. Для полного пути *p* через *h*(*p*) обозначим пару (начальный хост пути *p*, конечный хост пути *p*). Через *H*(*P*) обозначается множество пар хостов, связываемых путями из *P*: H(P) = {*h*(*p*) | *p* ∈ *P*}. Для этих обозначений пара (*Z*, *P*) является реализацией виртуальной сети (*Z*, *D*), если *H*(*P*) = *D*. Однако, как было показано в разделе 2, при реализации в коммутаторах правил, определяемых путями из *P*, могут порождаться дополнительные пути, и в общем случае эти правила порождают надмножество путей *P*↓↑ ⊇ *P*. Порождаемый путь *p* ∈ *P*↓↑ непредусмотренный, если *h*(*p*) ∉ *D*. Реализация *строгая*, если она не порождает непредусмотренных путей, т.е. *H*(*P*↓↑) = *D*.

Граф *G*, в котором любое нормальное множество пар хостов можно строго реализовать без циклов, называется *совершенным*.

Конечное множество *P* полных путей, связывающее все пары разных хостов, т.е. *H*(*P*) наибольшее нормальное множество пар хостов, называется *совершенным*, если в *P* нет двух путей вида ***p****ab****r*** и ***p****'ab****r****'*, где ***p*** ≠ ***p****'* и ***r*** ≠ ***r****'*. Очевидно, что *P* замкнуто по дугам, т.е. при замыкании по дугам не порождаются непредусмотренные пути.

**Proposition 14.** Граф, в котором есть совершенное множество *P* путей, совершенный. При этом множество *P* для каждого нормального множества пар хостов содержит его строгую реализацию без циклов как подмножество, а также без дублирования, если *P* не содержит дублирующих путей.

В [7] условие ослабляется: непредусмотренные пути допускаются, но только такие, которые не нарушают политику безопасности (security). Эта политика описывается в терминах приоритетов, присвоенных хостам, и формулируется в виде требования: приоритет хоста-получателя пакета должен быть не меньше приоритета хоста-отправителя пакетов. Пара хостов (*x*, *y*) называется *допустимой*, если приоритет *x* не больше приоритета *y*. Полный путь *p*, не нарушающий требования безопасности, т.е. такой, что пара *h*(*p*) допустима, называется *допустимым*. Реализация (*Z*, *P*) виртуальной сети (*Z*, *D*) называется *допустимой*, если замыкание по дугам *P*↓↑ содержит только допустимые пути. Граф *G*, в котором любое нормальное множество допустимых пар хостов имеет допустимую реализацию без циклов, называется *допустимо-совершенным*. Конечное множество *P* полных путей, связывающее все допустимые пары разных хостов, т.е. *H*(*P*) наибольшее нормальное множество допустимых пар хостов, называется *допустимо-совершенным*, если в *P* нет двух путей вида ***p****ab****r*** и ***p****'ab****r****'*, где ***p*** ≠ ***p****'* и ***r*** ≠ ***r****'*.

**Proposition 15.** Граф, в котором есть допустимо-совершенное множество *P* путей, допустимо-совершенный. При этом множество *P* для каждого допустимого нормального множества пар хостов содержит его строгую реализацию без циклов как подмножество, а также без дублирования, если *P* не содержит дублирующих путей.

В то же время, как показано в [5, 7], наличие в графе *G* совершенного (допустимо‑совершенного) множества путей является достаточным, но не является необходимым условием того, чтобы граф *G* был совершенным (допустимо‑совершенным).

Концепцию «совершенности» распространим на набор виртуальных сетей, заданный парой равномощных семейств: семейства множеств идентификаторов Z = { *Z*1, ..., *Zn* } и семейства нормальных множеств пар хостов D = { *D*1, ..., *Dn* }. При наличии приоритета хостов и основанной на них политики безопасности виртуальная сеть (*Zi*, *Di*) допустима, если допустимо множество пар хостов *Di*.

**Proposition 15.** Если в графе *G* есть совершенное (допустимо-совершенное) множество *P* путей, то на этом графе любой набор (Z, D) (допустимых) виртуальных сетей реализуется строго (допустимо) и без циклов. При этом множество *P* для каждого (допустимого) нормального множества *Di* пар хостов содержит его строгую (допустимую) реализацию *Pi* без циклов как подмножество *Pi* ⊆ *P*. Если *P* не содержит дублирующих путей, то *Pi* также не содержит дублирующих путей.

Пусть в графе *G* есть совершенное (допустимо-совершенное) множество *P* путей. Из propositions 14 и 15 следует, во-первых, то, что никакие два пути из *P* не порождают пути, отличные от них самих, и, во-вторых, то, что для любого (допустимого) нормального множества *D* пар хостов *P* содержит подмножество *PD* ⊆ *P* путей, соединяющих эти пары хостов, т.е. *H*(*PD*) = *D*. Тем самым, для каждой виртуальной сети (*Zi*, *Di*) можно выбрать реализацию (*Zi*, *Pi*), где *Pi* ⊆ *P*, которая является строгой (допустимой) и не порождает циклов. Если *P* не содержит дублирующих путей, то их не содержит и любое подмножество *Pi*.

1. **Заключение**

В статье предложены алгоритмы верификации реализации виртуальных сетей на плоскости данных SDN, целью которой является обнаружение таких нежелательных эффектов как непредусмотренные пути движения пакетов, циклические пути и дублирующие пути. Даже если каждая виртуальная сеть верифицирована отдельно и указанных эффектов не обнаружено, при совместной реализации нескольких виртуальных сетей эти эффекты могут возникнуть. Поэтому предложены две модификации алгоритма верификации реализации одной виртуальной сети для верификации совместной реализации нескольких виртуальных сетей. Все предложенные алгоритмы имеют полиномиальную сложность относительно исходных данных. Также рассмотрен вопрос о возможности реализации на плоскости данных SDN любого набора виртуальных сетей без нежелательных эффектов и установлено достаточное условие этого. Дальнейшая работа может двигаться в направлении исследования других эффектов (например, проблемы перегрузки сети) или considering other kinds of specifications for user requests.

**Литература**

1. Yevtushenko, N., Burdonov, I. B., Kossachev, A., L´opez, J., Kushik, N., and Zeghlache, D. (2018). Test derivation for the software defined networking platforms: Novel fault models and test completeness. In 2018 IEEE East-West Design & Test Symposium EWDTS, pages 1–6.

<http://burdonov.ru/doctor/papers_2018/Test_Derivation_for_the_Software_Defined_Networking_Platforms/Test_Derivation_for_the_Software_Defined_Networking_Platforms.pdf>

1. Burdonov I.B., Yevtushenko N.V., Kossatchev A.S. Testing switch rules in software defined networks. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 6, 2018, pp. 69-88. (In Russ.)

<http://burdonov.ru/doctor/papers_2018/Testirovaniye_pravil_nastroyki_setevogo_kommutatora/Testirovaniye_pravil_nastroyki_setevogo_kommutatora.pdf>

<https://ispranproceedings.elpub.ru/jour/article/view/1121>

1. Burdonov, I. B., Kossachev, A., Yevtushenko, N., L´opez, J., Kushik, N., and Zeghlache, D. (2019). Verifying SDN data path requests. CoRR, abs/1906.03101.

<https://arxiv.org/pdf/1906.03101v1.pdf>

1. Igor Burdonov, Nina Yevtushenko, Alexandr Kossachev. Implementing a Virtual Network on the SDN Data Plane. Proceedings 2020 IEEE East-West Design & Test Symposium (EWDTS). Varna, Bulgaria, September 4 – 7, 2020. pp. 279-283. ISBN: 978-1-7281-9898-9.

<http://burdonov.ru/doctor/papers_2020/Implementing%20a%20Virtual%20Network%20on%20the%20SDN%20Data%20Plane/Implementing%20a%20Virtual%20Network%20on%20the%20SDN%20Data%20Plane.pdf>

<https://publications.hse.ru/mirror/pubs/share/direct/397662745.pdf>

<https://ieeexplore.ieee.org/document/9224987>

1. Burdonov I.B., Vinarsky E.M., Yevtushenko N.V., Kossatchev A.S. Perfect sets of paths in the complete graph of SDN network switches. Proceedings of the Institute for System Programming of the RAS (Proceedings of ISP RAS). 2020;32(4):245-260. (In Russ.)

<http://burdonov.ru/doctor/papers_2020/Sovershennye%20mnozhestva%20putei%20v%20polnom%20grafe%20kommutatorov%20SDN-seti/Sovershennye%20mnozhestva%20putei%20v%20polnom%20grafe%20kommutatorov%20SDN-seti.pdf>

<https://ispranproceedings.elpub.ru/jour/article/view/1327>

1. Igor Burdonov, Alexandre Kossachev, Nina Yevtushenko, Jorge López, Natalia Kushik, Djamal Zeghlache. Preventive Model-based Verification and Repairing for SDN Requests. http://arxiv.org/abs/1906.03101, last revised September, 23, 2020.

<http://burdonov.ru/doctor/papers_2020/Preventive%20Model-based%20Verification%20and%20Repairing%20for%20SDN%20Requests/Preventive%20Model-based%20Verification%20and%20Repairing%20for%20SDN%20Requests.pdf>

<https://arxiv.org/abs/1906.03101>

1. Burdonov I.B., Yevtushenko N.V., Kossatchev A.S. Secure Implementing a Virtual Network on the SDN Data Plane. Proceedings of the Institute for System Programming of the RAS (Proceedings of ISP RAS). 2021;33(1):123-136. (In Russ.) https://doi.org/10.15514/ISPRAS-2021-33(1)-9

<https://ispranproceedings.elpub.ru/jour/article/view/1378>

1. Sperner, Emanuel (1928), "Ein Satz über Untermengen einer endlichen Menge", Mathematische Zeitschrift (in German), 27 (1): 544–548.

<https://zbmath.org/?format=complete&q=an:54.0090.06>

Нина, хотелось бы проверить и, если нужно, дополнить список литературы.

П. 3 «Verifying SDN data path requests» и п.6 «Preventive Model-based Verification and Repairing for SDN Requests» имеют почему-то одну и ту же идентификацию: 1906.03101.

И, наверное, нужна новая ссылка на ENASE 2021, я её у себя что-то не нашёл.

Можно было бы дать ещё ссылку, но этого номера пока нет. Нужно ли?

статья «Совершенные множества путей в полном графе коммутаторов SDN-сети», опубликованная в Трудах ИСП РАН, была выбрана для публикации в журнале "Programming and Computer Software" главным редактором журнала А.И. Аветисяном. Публикация состоится в 7 или 8 номерах 2021 (оба номера выходят в декабре 2021).

ССЫЛКИ НА ПРОЕКТЫ — с Петренко связаться