Министерство образования и науки Российской Федерации
Московский физико-технический институт
(национальный исследовательский университет)

Физтех-школа радиотехники и компьютерных технологий
Кафедра системного программирования (ИСП РАН)

Выпускная квалификационная работа бакалавра

Автоматическая генерация сигнатур сетевых протоколов и приложений

Автор:
Студент Б01-009а группы
Дурнов Алексей Николаевич

Научный руководитель:
канд. физ.-мат. наук
Гетьман Александр Игоревич

Москва 2024

Аннотация

Автоматическая генерация сигнатур сетевых протоколов и приложений

Дурнов Алексей Николаевич

Краткое описание задачи и основных результатов, мотивирующее прочитать весь текст

Содержание

1 Введение

Интернет-провайдеры и сетевые администраторы хотят идентифицировать тип сетевого трафика, который проходит через их сеть, для того, чтобы предоставлять своим клиентам лучший сервис, предлагая им высокое качество обслуживание (QoS), а также планировать свою инфраструктуру и управлять ею. Сетевой трафик можно классифицировать как в соответствии с использующимся протоколом, так и в соответствии с использующимся приложением. Вторая классификация более трудоёмкая, но позволяет решать более широкий спектр задач: формирование трафика в сети, улучшение качества обслуживание, а также предоставление более детального биллинга.

В области классификации сетевого трафика было проведено множество исследований, что привело к разработке многих методов. Самый наивный метод классификации сетевого трафика это идентификация по номеру порта. На заре развития интернета многие клиент-серверные приложения имели четко определенные номера портов сервера, которые определяются Internet Assigned Numbers Authority (IANA) [1]. К примеру, DNS использует номер порта сервера 53, SMTP использует номер порта сервера 25, HTTP - номер 80 и т.д. Таким образом, используя известные номера портов, можно классифицировать трафик. Однако современные приложения используют динамическое распределение портов и туннелирование трафика, например, по протоколу HTTP, поэтому этот метод на данный момент не обладает необходимой точностью [2]. Чтобы преодолеть эти ограничения идентификации были введены более совершенные методы [3].

В рамках подхода глубокого анализа пакетов (DPI) анализатор просматривает содержимое каждого пакета полностью. Эта технология получило очень широкое распространение, так как эти методы являются точными для идентификации трафика, однако DPI является трудоемким и ресурсоемким процессом. Ведущий подход, использующиеся в DPI для классификации трафика, основан на сопоставлении сигнатур полезной нагрузки. Под сигнатурой понимается часть данных полезной нагрузки, которая является статичной и различимой для приложений/протоколов и может быть описана, как последовательность строк или шестнадцатиричных чисел.

Подход, альтернативный DPI, основан на статистическом анализе. Статистический анализ реализуются в основном с помощью алгоритмов машинного обучения [4]. Для этого подхода используются уже косвенные признаки пакетов и потоков, такие как задержки между пакетами, размеры пакетов и другие. Существует несколько проблем при классификации сетевого трафика с использованием данного подхода. Во-первых, статистические характеристики, используемые при классификации, нестабильны, поскольку, например, задержка и коэффициент потери пакетов в сети динамичны. Во-вторых, не все потоки конкретного приложения имеют очевидные и особые характеристики трафика. Потоки, принадлежащие разным приложениям, могут иметь схожую статистику. Трудно отличить эти похожие потоки с помощью их свойств. Несмотря на эти проблемы, этот подход может применяться для зашифрованного сетевого трафика. Так как в первом приближении можно считать, что полезная нагрузка зашифрованного трафика представляет собой белый шум, поэтому сигнатурный подход не применим.

Чтобы точно классифицировать поток, система DPI должна искать сигнатуры в полезной нагрузке пакетов. Такой подход не нов, системы обнаружения вторжения (IDS) с помощью сигнатур находят интернет-червей и другой трафик, угрожающий безопасности [5–7]. По сравнению с проблемой генерации сигнатур червей проблема генерации сигнатур безопасностного трафика имеет несколько отличий. Во-первых, общие подстроки в прикладных протоколах обычно очень короткие, а некоторые составляют всего несколько байт, в то время как общие подстроки у червей обычно достигают несколько

десятков или сотен байт. Некоторые методы [5,6] оказываются неэффективными, когда общие подстроки короткие. Во-вторых, разные потоки в одном и том же приложении могут иметь разные общие подстроки. Например, в P2P приложении некоторые потоки используются для обмена одноранговой информацией, а другие потоки используются для обмена данными. Они могут использовать разные протоколы и иметь разные общие подстроки.

Поначалу сигнатуры извлекались вручную, но и сейчас часто пишутся в том числе на заказ [8]. Постоянное появление новых приложений и их частые обновления подчёркивают необходимость автоматической генерации сигнатур, так как ручная операция извлечения сигнатур занимает много времени, а также может быть разница в качестве сигнатур в зависимости от оператора извлечения.

Методы автоматической генерации сигнатур должны быть основаны не на семантическом анализе протоколов, так как, хотя они и повышают точность сигнатур, но не могут быть применены к анализу высокоскоростного трафика в режиме реального времени [9].

Поэтому данная работа посвещена исследованию различных методов автоматической генерации сигнатур полезной нагрузки для классификации сетевого трафика по протоколам и приложением.

2 Постановка задачи

Целью данной работы является разработка и реализация метода автоматической генерации сигнатур полезной нагрузки сетевого трафика для классификации этого трафика в соответствии с использующимся протоколом или приложением в режиме реального времени.

Для достижения поставленной цели необходимо решить следующие задачи:

1. Провести исследование литературы по соответствующей теме.
2. Собрать набор сетевых трасс для последующего тестирования и сравнения методов.
3. Выбрать оптимальный метод для автоматической генерации сигнатур.
   (a) Выбрать оптимальный набор параметров метода для каждого тестируемого протокола и приложения, если метод обладает настраиваемыми параметрами.
   (b) Найти ограничения рассматриваемых методов.
4. Разработать алгоритм генерации сигнатур на основе выбранного метода.
5. Разработать классификатор сетевого трафика для проверки сгенерированных сигнатур.
6. Встроить генератор сигнатур и классификатор как модули в систему анализа высокоскоростного сетевого трафика, разрабатываемую в ИСП РАН.

3 Обзор существующих решений

3.1 Формат сигнатур

Прежде чем говорить непосредственно о методах автоматической генерации сигнатур, стоит сначала понять, какие бывают сигнатуры и в каком формате они могут быть представлены.

Существует несколько представлений сигнатур. Некоторые из этих видов использовались для представлений сигнатур червей. В работе [7] приводится следующая классификация:

- **Конъюнктивная сигнатура**: состоит из набора подстрок (или токенов), полезная нагрузка соотвествует ей, если все токены в наборе были найдены в любом порядке.
- **Сигнатура, представленная последовательностью токенов**: состоит из упорядоченного набора токенов. Полезная нагрузка соотвествует сигнатуре, если она содержит всю последовательность токенов в том же порядке.
- **Байесовская сигнатура**: состоит из набора токенов, каждый из которых связан с оценкой, и общего порогового значения. Байесовские сигнатуры обеспечивают вероятностное сопоставление: вычисляется вероятность сопоставления, используя оценки присутствующих токенов в полезной нагрузки. Если результирующая вероятность превышает пороговое значение, то считается, что полезная нагрузка совпала с сигнатурой.

При использовании любого из этих определений в качестве определения сигнатуры приложения/протокола возникают некоторые специфические проблемы:

1. **Разнообразие протоколов приложения**: два потока, принадлежащие одному и тому же приложению, могут иметь разные общие подстроки. Например, приложения P2P используют разные протоколы для обмена одноранговой информацией и данными. Кроме того приложения будут обновлять свои протоколы. Потоки различных версий протоколов могут существовать в сети одновременно. Ни конъюктивная сигнатура, ни сигнатура, представленная токен-подпоследовательностью, не могут выражать несколько протоколов.
2. **Взаимоисключающее свойство некоторых подстрок в прикладных протоколах**. Например, протокол Gnutella имеет две последовательности подстрок {'Get' 'UserAgent'} и {'HTTP' 'User-Agent'}. Подстроки 'Get' и 'HTTP' не будут отображаться в одном и том же потоке в протоколе Gnutella. Две общие подстроки являются взаимоисключающими, но байесовские сигнатуры не могут выражать взаимоисключающее свойство.

Определение сигнатуры, основанное на регулярных выражениях, становится очень распространённым в классификации приложений [10–12]. Однако процесс сопоставления регулярных выражений требует огромной вычислительной мощности, которая слабо масштабируется для идентификации сетевого трафика в режиме реального времени. Способ построения регулярного выражения оказывает непосредственное влияние на классификацию потоков и на общую производительность сопоставления. Несмотря на это, некоторые системы DPI используют регулярные выражения для представления сигнатур приложений. Система обнаружения/предотвращения вторжений Snort

(IDS/IPS) [13] имеет множество сигнатур приложений и предлагает пользователю возможность вставлять новые регулярные выражения по требованию.

Почти неизвестно алгоритмов извлечения сигнатур в виде регулярных выражений, а те что есть - основаны на строках и небольшом подмножестве операций, которые есть в регулярных выражениях. Поэтому дальше будем рассматривать сигнатуры только в виде строк.

Среди возможных описаний строковых сигнатур, набор последовательностей подстрок является наилучшим определением сигнатуры. Полезная нагрузка соотвествует сигнатуре, если она содержит какую-то последовательность подстрок из этого набора. Такое определение решает описанные выше проблемы.

Введем следующее определение: уровень поддержки последовательности подстрок (support) равен отношению количества хостов, использующих соответствующее приложение или протокол и трафик которого содержит эту последовательность подстрок к общему количеству хостов, использующих соответствующее приложение или протокол.

При заданном формате, в общем случае, не любая последовательность подстрок из набора охватывает все хосты. В части из дальнейших рассматриваемых методов считается, что сигнатура состоит из одной последовательности подстрок, уровень поддержки которой равен 1, то есть она присутствует в трафике каждого хоста, использующего рассматриваемый протокол или приложение. Этот показатель можно использовать как параметр в некоторых из этих методов.

3.2 Структура сигнатур

Большинство форматов сигнатур в предыдущих работах [9, 14, 15] представляют собой простые подстроки, которые часто появляются в полезной нагрузке. Следовательно, всё ещё существует вероятность того, что извлеченные сигнатуры полезной нагрузки могут быть не специфичными для конкретного приложения, некоторые могут принадлежать и другому приложению. Это называется избыточностью сигнатур.

Для улучшения качества сигнатур предлагается выделить три типа сигнатур [16, 17]:

1. сигнатура содержимого (полезной нагрузки),
2. сигнатура пакета,
3. сигнатура потока.

Сигнатура содержимого определяется как различимая и уникальная подстрока полезной нагрузки, состоящая из непрерывных символов или шестнадцатеричных значений. На самом деле уникальность с помощью одной подстроки тяжело обеспечить, например, такие строки “GET” или “HTTP”, которые часто встречаются в HTTP, не могут служить конечными сигнатурами, так как они не различают приложения.

Сигнатура пакета состоит из серии сигнатур содержимого, которые появляются в одном пакете. Так как классификация может выполняться без накопления пакетов, то есть без сбора потока, то анализируется всегда хотя бы один пакет. Это значит, что для классификации всегда можно использовать сигнатуры пакетов, и не имеет смысла использовать отдельно сигнатуры содержимого.

Сигнатура потока состоит из серии сигнатур пакетов, которые появляются в одном потоке, где под потоком понимается набор пакетов, имеющий одни и те же IP-адрес источника, IP-адрес назначения, порт источника, порт назначения и используемый протокол транспортного уровня. Сигнатура потока гораздо более специфична для конкретного приложения, чем сигнатура пакета, и значительно повышает точность.

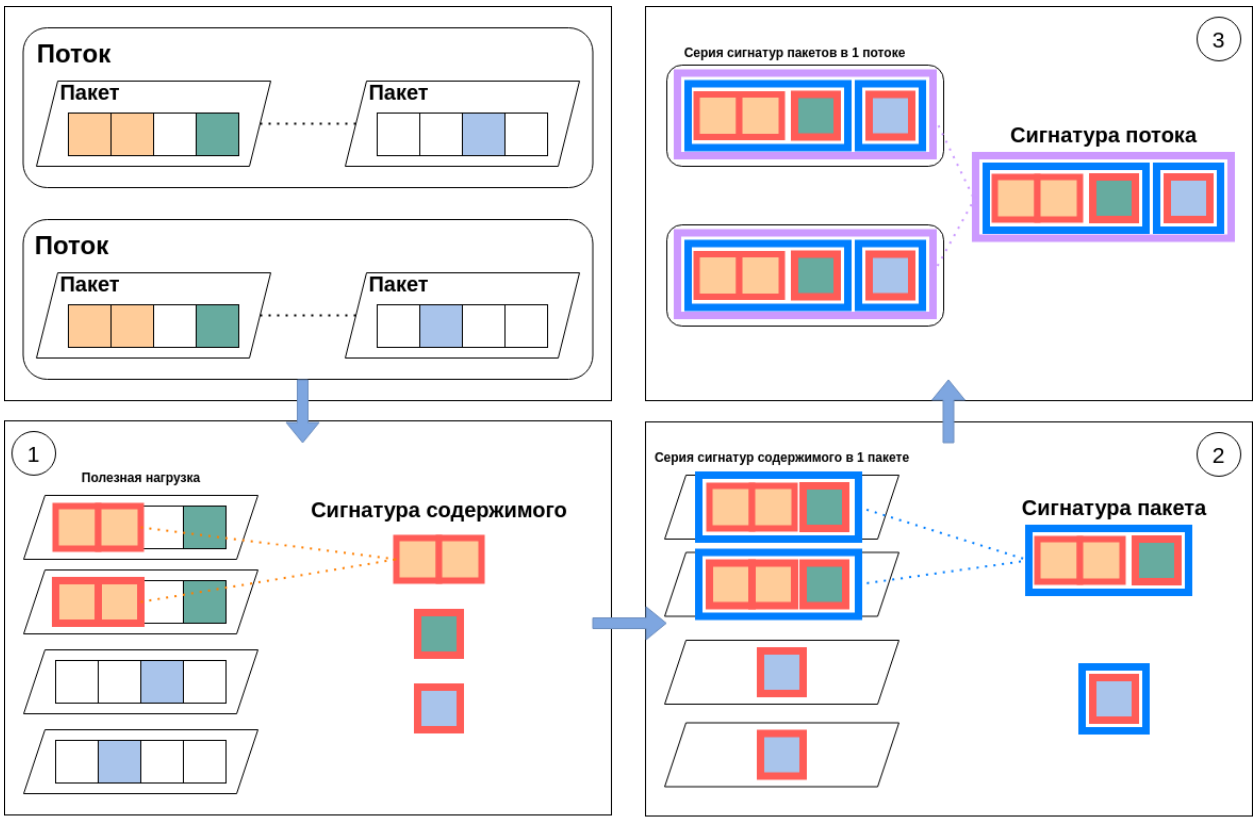

Рис. 1: Процесс извлечения предлагаемой структуры сигнатур полезной нагрузки.

Поэтапный процесс извлечения сигнатур представлен на рис. 1. Как можно заметить, данная структура сигнатур обладает свойством вложенности.

3.3 Метрики оценки качества сигнатур

Для оценки качества получаемых сигнатур рассмотрим матрицу ошибок: 4 стандартные категории, к котором можно отнести результат работы классификатора на полученной сигнатуре. В нашем случае рассматриваемый класс это целевой протокол или приложение. Под классификацией трафика будем понимать классификацию конкретного пакета или потока в зависимости от того, какой уровень сигнатур используется.

	Принадлежит классу (P)	Не принадлежит классу (N)
Предсказана принадлежность к классу (T)	TP	TN
Предсказано отсутствие принадлежности к классу (F)	FP	FN

- истинно положительный (TP): указывает, что трафик правильно классифицирован, как относящийся к определенному классу.
- истинно отрицательный (TN): указывает, что трафик правильно классифицирован, как не относящийся к определенному классу.
- ложно положительный (FP): указывает, что трафик неправильно классифицирован, как относящийся к определенному классу.

- ложный отрицательный (FN): указывает, что трафик неправильно классифицирован, как не относящийся к определенному классу.

Наиболее часто используемые показатели для классификации трафика определяются следующим образом:

- Accuracy (достоверность) - доля правильных классификаций.

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

- Recall (полнота) - отношение верно классифицированного трафика определенному классу к общему числу трафика этого класса, то есть описывает способность сигнатуры обнаружить данный целевой протокол или приложение.

$$recall = \frac{TP}{TP + FN}$$

- Precision (точность) - доля верно классифицированного трафика среди всего трафика, который классификатор отнёс к этому классу.

$$precision = \frac{TP}{TP + FP}$$

- F_1-score (F_1-мера) - гармоническое среднее между точностью и полнотой.

$$F_1\text{-}score = \frac{2 \times recall \times precision}{recall + precision}$$

Метрика accuracy может терять свой смысл в задачах с сильно неравными классами. Напротив же recall и precision не зависят от соотношения классов и поэтому применимы в случае несбалансированных классов, что является правдой для сетевого трафика. Часто на практике возникает задача найти оптимальный баланс между precision и recall. Для этих целей подходит F-мера, которая достигает максимума при recall и precision равным 1, и стремится к минимуму, если хотя бы один из параметров стремится к нулю.

Для сигнатур полезно ещё ввести такое понятие как:

- Redundancy (избыточность) опредялется как:

$$Redundancy = \frac{\text{Объём трафика идентифицированный двумя и более сигнатурами}}{\text{Объём трафика идентифицированый набором сигнатур}}$$

Redundancy имеет значение от 0 до 1, где 0 - наилучшее значение, которое указывает на то, что все сигнатуры набора классифицируют исключительно только свою часть трафика, то есть являются уникальными и незаменяемыми. Если redundancy близка к 1, то в наборе присутствует ненужные сигнатуры, которые идентифицируют перекрывающийся трафик. По мере увеличения количества сигнатур увеличиваются и накладные расходы системы, поэтому данное значение должно оставаться низким.

3.4 Обзор существующих методов автоматической генерации сигнатур

Рассмотрим несколько существующих методов автоматической генерации сигнатур и опишем основной принцип их работы.

3.4.1 LASER

В статье [9] описан алгоритм LASER (Application Signature ExtRaction), который основан на задаче поиска наиболее длинной общей подпоследовательности LCS (Longest common subsequence). Данный алгоритм автоматически определяет достоверный шаблон в полезной нагрузке пакета без предварительного знания форматов протоколов, то есть генерирует сигнатуру пакета. За шаг извлечения сигнатуры был принят алгоритм LCS, который ранее в основном использовался для сопоставления последовательностей ДНК в приложениях биоинформатики [18]. Он был модифицирован под текущие задачи.

Вводятся ограничения на модификацию LCS:

- **Количество пакетов в потоке**. Нет необходимости проводить проверку над всеми пакетами в наборе, потому что сигнатура существует в нескольких начальных пакетах потока.
- **Минимальная длина подстроки**. Стоимость сопоставления сигнатур пропорциональна их длинам. Сгенерированная сигнатура состоит из последовательности подстрок. Чтобы избежать каких-либо тривиальных сигнатур, минимальная граница длины подстроки должна рассматриваться как ограничение для модифицированного алгоритма LCS. Имея это ограничение длины, предотвращается включение однопозиционных и многопозиционных символов в последовательность общих строк, например символа '/' в HTTP пакетах.
- **Сравнение размера пакетов**. Это увеличивает вероятность нахождения надёжной сигнатуры, если пакеты сгруппированы по назначению (например, управляющий трафик или трафик загрузки) и характеристикам трафика. Одной из таких характеристик является размер пакета. Обьём данных при установлении соединения небольшой. Конкретная сигнатура существует только в первых пакетах. Поэтому сравнения небольших пакетов установки соединения и пакетов загрузки нежелательно для генерации надёжной сигнатуры.

Приведём сам алгоритм из статьи:

Листинг 1: Алгоритм LASER

```
procedure Signature_Generation ()
    Flow_Pool {F1[]...Fx[]} <- Santized_packet_collector
    F1[] <- Iterate, packet dump for Flow 1
    F2[] <- Iterate, packet dump for Flow 2
    while i from 0 to packet_constraint do
        while j from 0 to packet_constraint do
            if |F1[i].packet_size - F2[j].packet_size| < threshold
            result_LCS <- LASER (F1[i], F1[j])
            LCS_Pool{} <- Append result_LCS, end if
        j++, end while
    i++, end while
    S <- select the longest from LCS_Pool
```

```
13      while i from 0 to number of rest flows of Flow_Pool do
14          Fi <- select one from the rest of Flow_Pool
15          result_LCS <- LASER (S, Fi)
16          S <- select the longest from result_LCS
17      i++, end while
18      return S
19
20  procedure LASER (PacketA[1...m], PacketB[1...n])
21      PacketA [m...1] <- Reverse byte stream
22      PacketB [n...1] <- Reverse byte stream
23      Matrix [m][n]
24      while i from 0 to m do
25          while j from 0 to n do
26              if i = 0 or j = 0, then Matrix[i][j] = 0;
27              else if PacketA[i] = PacketB[j], then
28                  Matrix [i][j] <- 'Diagonal',
29                  Matrix [i][j] = Matrix [i-1][j-1] + 1;
30              else if Matrix[i-1][j] >= Matrix[i][j-1], then
31                  Matrix[i][j] <- 'Up',
32                  Matrix[i][j] = Matrix [i-1][j];
33              else
34                  Matrix[i][j] <- 'Up',
35                  Matrix[i][j] = Matrix [i][j-1];
36          end while
37      end while
38      i <- m-1; j <- n-1 /* Tracking */
39      while Matrix[i][j] != 0 do
40          if Matrix[i][j] = 'Left', then
41              j--
42          else if Matrix[i][j] = 'Up', then
43              i--
44          else if Matrix[i][j] = 'Diagonal', then do
45              Substring <- Append PacketA[i]
46              if Matrix[i-1][j-1] != 'Diagonal', then
47                  Substring <- Append special break point character (e.g. '/')
48      i--; j--, end while
49      while tokenizing substring based on break point do
50          if token_length > minimum_substring_length_constraint
51              then, result_LCS <- Append token_substring
52      end while
53      return result_LCS
```

Поясним работу алгоритма. К моменту генерации сигнатуры у нас есть набор потоков, в которых содежится хотя бы по packet_constraint (параметр нашего алгоритма) пакетов (строка 2). Дальше из них произвольно выбираются два потока (строки 3-4). Затем происходит полный попарный перебор пакетов из двух потоков, и в случае, если разница размеров пакетов в паре меньше threshold (параметр нашего алгоритма), то вызывается функция LASER для этой пары пакетов (поиск наиболее длинной общей подпоследовательности), а результат работы заносится в пул (строки 5-11). Из этого пула выбирается самая длинная общая подпоследовательность (строка 12). Затем происходит процесс уточнения: проходим по всем оставшимся потокам, вызываем функцию LASER для всех пакетов из выбранного потока и текущей сигнатуры, среди результатов выбирается самая длинная общая подпоследовательность (строки 13-17). Когда потоки

в пуле закончились, возвращаем полученный результат (строка 18).

Функция LASER представляет собой алгоритм Needleman-Wunsch, в котором используется матрица направлений. Так как пакеты развернуты (строки 21-22), то на самом деле матрица заполняется с конца (строки 23-37). Затем обходится в обратном порядке, по направлениям и собирается общая подпоследовательность (строки 38-48). Дальше из общей подпоследовательности, состоящая из токенов, выкидываются те, что короче minimum_substring_length_constraint (параметр нашего алгоритма) (строки 49-53). И возвращаем полученный результат (строка 50).

Таким образом, у алгоритма LASER 3 параметра настройки:

- packet_constraint (количество пакетов в потоке)
- threshold (порог разницы размеров для сравнения пакетов)
- minimum_substring_length_constraint (минимальная длина подстроки)

3.4.2 AutoSig

Следующий рассматриваемый метод называется AutoSig [14, 15]. Каждый поток из выборки обрабатывается как последовательность байт из первых N байт полезной нагрузки. Данный алгоритм генерирует сигнатуру-потока, являющуюся набором общих последовательностей подстрок. Приведем описание работы алгоритма.

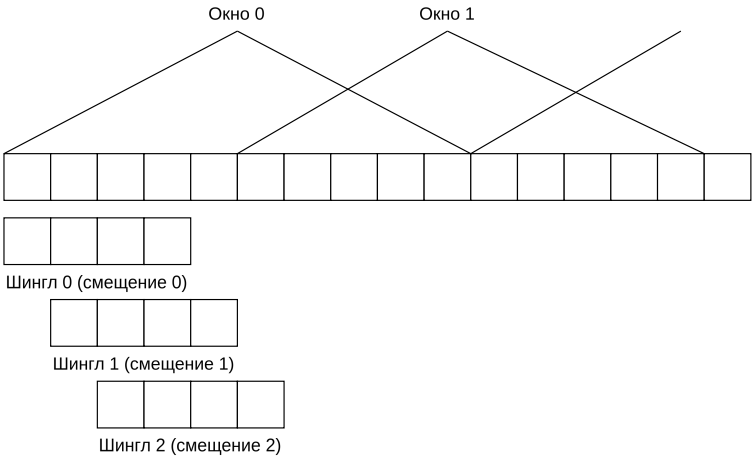

Рис. 2: Схема выделение шинглов и окон в алгоритме AutoSig.

Сначала алгоритм делит содержимое потоков на небольшие блоки содержимого фиксированного размера, которые называются шинглами. Чтобы исключить шумы, шинглы делят на разные группы в зависимости от их смещения. Как видно из рис. 2 шинглы разделены на разные окна. Окна перекрываются и имеют фиксированную ширину 2W. i-е окно покрывает полезную нагрузку, которая расположена от $i \times W$ байта до $(i + 2) \times W$ байта, поэтому размер области перекрытия между двумя окнами равен размеру W. Шингл будет считаться общим, если он встретился в одном и том же окне более, чем в $R \cdot N$ потоках, где N - количество потоков, R - порог выбора общего шингла. Затем при помощи адаптивного алгоритма слияния перекрывающиеся общие шинглы объединяются в подстроки. Для оценки их слияния используется показатель

$$sim(x, y) = \frac{|flows(xy)|}{|flows(x) \cup flows(y)|}$$

где $flows(x)$ - потоки, содержащие шингл x, а $flows(xy)$ - потоки, содержащие шинглы x и y вместе (смежные или перекрывающиеся, иначе $sim(x, y) = 0$). В алгоритме адаптивного слияния шингл x и шингл y объединяются, если $sim(x, y) > S$, где S - предопределеное пороговое значение. Наконец, строится дерево подстрок для организации общих подстрок в конечные сигнатуры.

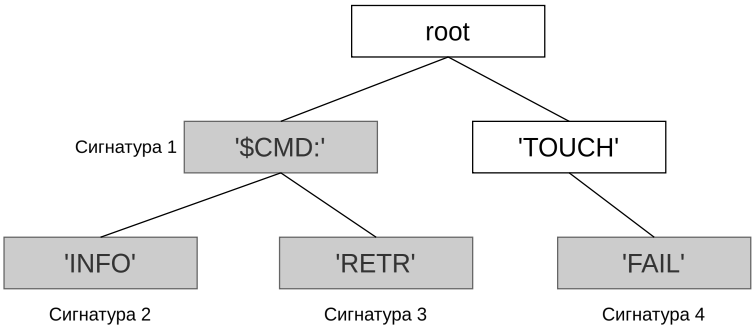

Рис. 3: Дерево подстрок в алгоритме AutoSig.

Серые узлы являются узлами сигнатуры. Корневой узел дерева подстрок является пустой подстрокой. Каждый путь от узла сигнатуры к корневому узлу соответствует последовательности подстрок. Он формируется подстроками в узлах вдоль пути (исключая корневой узел). Дерево подстрок обладает следующими свойствами:

- Каждый узел соответствует общей подстроке и набору потоков F. Каждый поток в этом наборе содержит общую подстроку. Общая подстрока корневого узла является пустой строкой, будем считать, что она содержится во всех потоках.
- если узел P является родительским узлом узла A, набор потоков F(A) является подмножеством набора потоков F(P). Это подразумевает, что поток, принадлежащий узлу A, не только содержит общую подстроку узла A, но также содержит общую подстроку своего родительского узла P. Например, потоки узла ‘INFO’ содержат подстроки ‘INFO’ и ‘$CMD:’.
- Если набор потоков узла является строгим надмножеством объединения наборов потоков его дочерних узлов, узел является узлом сигнатуры. В противном случае это не узел сигнатуры. Корневой узел не является узлом сигнатуры. Листья являются узлами сигнатуры. Например, на рис. 3 узел ‘FAIL’ является узлом сигнатуры, а узел ‘TOUCH’ - нет. Потоки, содержащие подстроку ‘TOUCH’, также содержат подстроку ‘FAIL’. Вместо двух последовательностей подстрок {‘TOUCH’, ‘FAIL’} и {‘TOUCH’} генерируется только последовательность подстрок {‘TOUCH’, ‘FAIL’}.

Листинг 2: Алгоритм построения дерева подстрок

```
constructTree(substrings)
  Node root = new Node('');
  Tree tree = new Tree(root);
  Sort(substrings);
  for (int i =0; i< substrings.length ; i++)
    addNode(root, new Node(substrings[i]));

boolean addNode(curNode, newNode )
```

```
9   if (!isSubFlowSet(curNode, newNode))
10    return false;
11  boolean succ = false;
12  for every child node of curNode
13    if (addNode(child, newNode))
14      succ = true;
15  if (!succ)
16    node.addChild(newNode);
17  return true
```

Сначала строится дерево только с пустым корнем (строки 2-3). Затем общие подстроки сортируются в порядке убывания в соответствии с количеством потоков, в которых они присутствуют (строка 4). Потом эти подстроки вставляются в дерево одна за другой (строки 5-6).

Функция AddNode используется для вставки нового узла подстроки. Сначала она проверяет, является ли набор потоков нового узла подстроки подмножеством набора потоков текущего узла (строка 9). Если это не так, функция напрямую возвращает ошибку (строка 10). В противном случае, пытаемся вставить новый узел подстроки под каждым дочерним узлом текущего узла, вызывая функцию рекурсивно (строки 11-14). Если все вставки завершатся неудачей, новый узел подстроки будет вставлен как ребенок текущего узла (строки 15-16).

Каждый путь от узла сигнатуры к корневому узлу соответствует последовательности подстрок. Для генерации последовательностей подстрок используется алгоритм поиска в глубину со стеком для записи пути. После извлечения последовательностей подстрок в дереве каждая последовательность подстрок переупорядочивается в соответствии со смещением в потоке каждой подстроки.

3.4.3 SigBox

Теперь рассмотрим метод, который называется SigBox [19]. Данный метод основан на алгоритме последовательных шаблонов. Конечная цель этого алгоритма состоит в том, чтобы найти частые подпоследовательности в наборе входных последовательностей. Применяется один и тот же алгоритм для извлечения трех типов сигнатур: содержимого, пакета и потока. Изменяется лишь то, какие последовательности подаются на вход алгоритму и чем является элемент этой последовательности.

В случае извлечения сигнатуры содержимого последовательность содержимого является полезной нагрузкой пакета. Следовательно, элемент последовательности представляет собой однобайтовый символ (его шестнадцатеричное значение) полезной нагрузки пакета.

В случае извлечения сигнатуры пакета последовательность пакетов представляет собой серию сигнатур содержимого, находящаяся в одной и той же полезной нагрузки пакета. Следовательно, элемент последовательности пакетов представляет собой индивидуальную сигнатуру содержимого.

В случае извлечения сигнатуры потока последовательность потока представляет собой серию сигнатур пакетов, расположенных в одной и той же полезной нагрузки потока. Следовательно, элемент последовательности потока представляет собой индивидуальную сигнатуру пакета.

Листинг 3: Алгоритм выделения подпоследовательности

```
1 subsequenceExtractor(SequenceSet, MinSupp):
2   for each sequence S in SequenceSet do
3     for each item i in sequence S do
```

```
4       L_1 ← L_1 ∪ i;
5     end
6   end
7   k ← 2
8   while L_{k−1} ≠ ∅ do
9     for each candidate c in L_{k−1} do
10      supp ← calSupport(c, SequenceSet);
11      if (supp < Minsupp) then
12        L_{k−1} ← L_{k−1} - c;
13      end
14    end
15    L_k ← extractCandidate(L_{k−1});
16    k++;
17  end
18  SubSequenceSet ← ∪_k L_k;
19  deleteSubset(SubSequenceSet);
20  return SubSequenceSet;
```

Опишем работу данного алгоритма. Сначала извлекаются подпоследовательности длиной 1 из всех последовательностей и сохраняем их в наборе подпоследовательностей длиной 1, L_1 (строки 2-6). Из подпоследовательностей длиной 1 извлекаются все подпоследовательности кандидаты длины k, увеличивая длину до тех пор, пока не будут извлечены новые подпоследовательности (кандидаты) (строки: 7-17). Этот итерационный процесс состоит из двух частей. Сначала исключаются кандидаты, которые не удовлетворяют минимальной поддержке (1.0) после получения значения поддержки из функции calSupport (строки 9-14). Затем извлекаются кандидаты длиной k с помощью кандидатов длиной $(k-1)$ (строки 15). В качестве последнего шага проверяется связь включения между подпоследовательностями; если связь найдена, включенные подпоследовательности удаляются (строка 19).

Листинг 4: Алгоритм вычисления поддержки

```
1  calSupport(candidate, SequenceSet):
2    for each sequence S in SequenceSet do
3      totalhost ← totalhost ∪ S.host_id;
4      for k = 1 to size of (S, < I_1 I_2 I_3 ... I_n >) do
5        p ← k, q ← 1;
6        while (S, < I_p > == candidate.< I_q >) do
7          p++, q++;
8        end
9        if (q == size of (candidate.< I_1 I_2 I_3 ... I_n >)) then
10         supphost ← supphost ∪ S.host_id;
11         break
12       end
13     end
14   end
15   return supphost / totalhost;
```

Этот алгоритм получает на вход подпоследовательность и набор последовательностей и выводит поддержку данной подпоследовательности. Сначала сохраняются идентификаторы хостов всех последовательностей (строка 3). Затем алгоритм проверяет, включена ли подпоследовательность в последовательность для всех последовательностей во входном наборе. Если последовательность включает в себя подпоследовательность, она сохраняет соответствующий идентификатор хоста (строки 4-13). Наконец, этот алгоритм возвращает значение поддержки, вычисленное путем деления количе-

ства узлов поддержки на общее количество узлов (строка 15). В данном примере рассматривался наивный алгоритм сопоставления, однако авторы советовали заменить на более быстрые алгоритмы [20–23].

Листинг 5: Алгоритм извлечения кандидата

```
1  extractCandidate(L_{k-1})
2    for each candidate x in L_{k-1} do
3      for each candidate y in L_{k-1} do
4        if (k == 2) then
5          L_k ← L_k ∪ < x.< i_1 > y.< i_1 >>;
6        end
7        else if ((x.< i_2 > == y.< i_1 >) && (x.< i_3 > == y.< i_2 >) && ...
8                  (x.< i_{k-2} > == y.< i_{k-1} >)) then
9          L_k ← L_k ∪ < x.< i_1, i_2, i_3, ..., i_{k-1} > y.< i_{k-1} >;
10       end
11     end
12   end
13   return L_k;
```

Набор подпоследовательностей длины $(k-1)$, предоставленный в качестве входных данных, используется для извлечения набора подпоследовательностей длины k, L_k. Сравниваются все возможные пары L_{k-1} (строки 2-11). Если длина $(k-1)$ входной подпоследовательности равна 1, мы просто объединяем две подпоследовательности как подпоследовательность длиной 2 (строки 4-6). В противном случае мы сравниваем две подпоследовательности, чтобы выяснить, имеют ли эти две подпоследовательности объединяемый общий элемент. Если одна подпоследовательность, исключает его первый элемент, тогда он совпадает с другой подпоследовательностью, тогда как исключение его последнего элемента означает, что две подпоследовательности длины $(k-1)$ объединяются, образуя единую подпоследовательность длины k (строки 7-9).

Предложенная система генерирует сигнатуры содержимого, затем генерирует сигнатуры пакетов и, наконец, генерирует сигнатуры потоков, используя на каждом этапе описанный выше алгоритм.

4 Исследование и построение решения задачи

4.1 Сбор данных для дальнейшего тестирования

В данной работе генерация сигнатуры приложений рассматриваться не будет, так как большинство современные приложения используют шифрование, а административные методы, позволяющие дешифровывать поступающий трафик, остаются за рамками данного исследования. Однако были выбраны такие протоколы, которые покрывают все основные возможные проблемы, возникающие при генерации сигнатур приложений, которые были описаны раннее.

Будут рассматриваться следующие протоколы: HTTP, FTP, DNS, IMAP, SMTP, POP3, BitTorrent [24]. HTTP - типичный представитель текстового протокола, который используется не только веб-приложениями, но и в качестве туннеля. FTP использует множественное подключение (как минимум двойное), при этом один канал является управляющим, а через остальные происходит передача данных. DNS является представителем бинарного протокола, использующий UDP. IMAP, SMTP, POP3 - стандартные почтовые протоколы, сообщения у которых обычно короткие. BitTorrent - P2P протокол для кооперативного обмена файлами.

Исследуемый сетевой трафик снимался с кампусной сети ИСП РАН. Затем с помощью Wireshark [25] этот трафик был разбит по протоколам и результатом его работы были .pcap - файлы, которые содержали в себе сессии определенного протокола, захваченные в течение исследуемого сетевого взаимодействия. Пример выходного .pcap файла для HTTP можно увидеть на рис. 4.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	37.29.40.247	83.149.199.97	HTTP	1008	GET /api/tasks HTTP/1.1
2	0.017280	83.149.199.97	37.29.40.247	TCP	935	80 → 60785 [PSH, ACK] Seq=1 Ack=951 Win=1452 Len=877 [TCP seg…
3	0.017352	83.149.199.97	37.29.40.247	HTTP	78	HTTP/1.1 200 OK (application/json)
4	2.342100	37.29.40.247	83.149.199.97	HTTP	1008	GET /api/tasks HTTP/1.1
5	2.360254	83.149.199.97	37.29.40.247	TCP	935	80 → 60785 [PSH, ACK] Seq=898 Ack=1901 Win=1452 Len=877 [TCP …
6	2.360306	83.149.199.97	37.29.40.247	HTTP	78	HTTP/1.1 200 OK (application/json)
7	4.731904	37.29.40.247	83.149.199.97	HTTP	1008	GET /api/tasks HTTP/1.1
8	4.748144	83.149.199.97	37.29.40.247	TCP	935	80 → 60785 [PSH, ACK] Seq=1795 Ack=2851 Win=1452 Len=877 [TCP…
9	4.748264	83.149.199.97	37.29.40.247	HTTP	78	HTTP/1.1 200 OK (application/json)
10	7.039584	37.29.40.247	83.149.199.97	HTTP	1008	GET /api/tasks HTTP/1.1
11	7.056378	83.149.199.97	37.29.40.247	TCP	935	80 → 60785 [PSH, ACK] Seq=2692 Ack=3801 Win=1452 Len=877 [TCP…
12	7.056378	83.149.199.97	37.29.40.247	HTTP	78	HTTP/1.1 200 OK (application/json)

Рис. 4: Пример работы Wireshark.

4.2 Выбор алгоритма для автоматической генерации сигнатур

Алгоритм LASER достаточно простой, так как основан на LCS, поэтому отлично подходит для первичной разработки сопутствующей инфраструктуры, а также для различных модификаций, чтобы оценить поведения сигнатур в сетевом трафике при разных условиях.

Система AutoSig ввела дерево подстрок. Данная структура хороша тем, что увеличивает мощность получаемых сигнатур, это позволяет выделить несколько последовательностей подстрок, которые могут охватить те ситуация, когда приложение имеет сильно разные потоки (например, поток управления и поток данных в FTP). Однако в том в виде, в котором это дерево преставлено в работе, получается сильно избыточный результат. Если узел является сигнатурным и не листом, то по свойству этого узла его набор потоков является строгим надмножеством наборов потоков любых других сигнатурных узлов, находящихся в его поддереве (например, листы, которые всегда являются сигнатурными), таким образом, любые пути из сигнатурных узлов поддерева являются избыточнымы, так как если нашлась последовательность подстрок соответствующая

сигнатурному узлу из рассматриваемого поддерева, то найдётся в потоке и последовательность подстрок, соответствующая рассматриваемому узлу. При этом специфичность нашей сигнатуры за счёт этих сигнатурных узлов не увеличивается, так как для совпадения со сигнатурой достаточно совпадания последовательности подстрок, соответствующей нашему узлу, которая уже включена в другие. А значит поддерево можно удалить. Однако если не выполняется строгость надмножества, то последовательности строк поддерева не будут включаться друг в друга, а полнота покрытия потоков сохраниться, при этом вырастит специфичность (чем длиннее последовательность подстрок, тем специфичнее сигнатура). Именно поэтому при таком условии узел не становится сигнатурным.

SigBox должен быть не чувствителен к шумам, поэтому можно попробовать выделить сигнатуру на неразмеченном трафике.

4.3 Реализация алгоритма LASER

5 Описание практической части

5.1 Формат хранения сигнатуры

Выбранный формат сигнатуры - это набор последовательности подстрок. Представим требования хранения выбранного формата сигнатур в памяти:

1. Результатом работы оригинального алгоритма LASER является одна последовательность подстрок. Предполагается, что подстроки отделены специальным символом, но так как в бинарных протоколах используются все значения байта, то нельзя найти такой символ. Однако предположим, что такой символ найдется и заметим, что в момент уточнения сигнатуры, алгоритм LCS, который лежит в основе LASER, не чувствителен к наличию этого символа во входном потоке байт, так как это специальный символ и во втором потоке байт его нет. Значит последовательность подстрок необходимо хранить последовательно без разделяющего символа. Это также уменьшит размер матрицы направлений.

2. Для того, чтобы иметь доступ к самим подстрокам, например, с целью напечатать их, будем хранить смещения на каждую подстроку.

3. Для того, чтобы постоянно не расчитывать общую длину последовательности подстрок, будем хранить этот размер.

4. Необходимо также хранить и само количество подстрок, чтобы опредилить количество смещений.

5. Так как строка в `C/C++` должна быть нуль-терменированна, то в конце всех подстрок будет стоять ‘\0’. Сами подстроки дополнительно на ‘\0’ не оканчиваются.

6. Так же для быстрого доступа к подстроке и быстрого расчёта её длины, будем ещё будем хранить одно дополнительное смещение на ‘\0’, то есть суммарно $n+1$ смещение, где n - количество подстрок. Каждое смещение расчитывается относительно начала последовательности подстрок. Тогда размер подстроки можно легко вычислить через разницу смещений. Первое смещение всегда равно 0, его можно не хранить, но хранится для однородности вычислений.

7. Так как размер последовательности подстрок небольшой, то такое представление последовательности подстрок должно быть cache-friendly. Значит для представления набора последовательностей подстрок каждая последовательность должна быть локализирована.

8. Естественно надо хранить количество последовательностей и их смещения для доступа к ним.

Таким образом, данным требованиям удовлетворяет следующее представление:

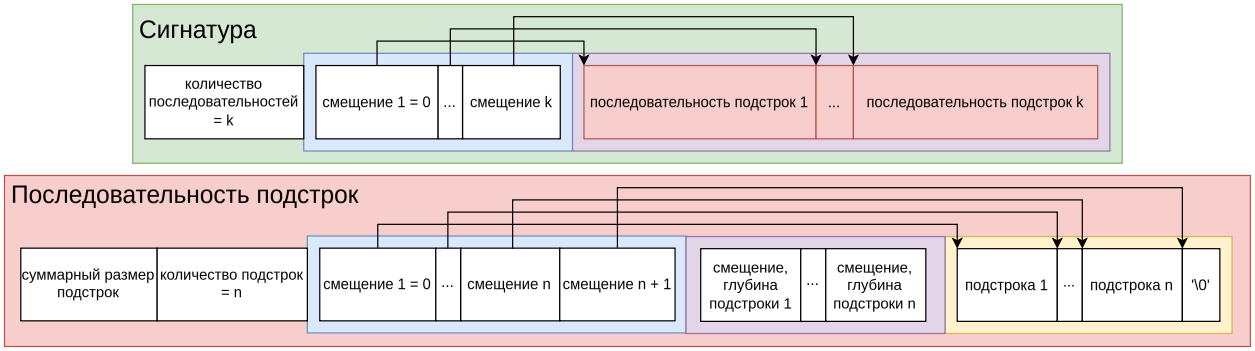

Рис. 5: Формат хранения сигнатуры.

Были добавлены специальные поля для каждой подстроки: смещение и глубина. Информация о смещении указывает начальную позицию для поиска подстроки в полезной нагрузке пакета, а информация о глубине указывает, как далеко поиск должен продолжаться от местоположения смещения. Если этой информации нет, то значения данных полей соответственно равны 0 и -1. Глубина равная -1 означает, что поиск требуется осуществлять до конца полезной нагрузки. Данные поля помогают ускорить процесс поиска подстрок при сопоставлении сигнатуры.

5.2 Интеграция в архитектуру системы анализа трафика

Система анализа высокоскоростного трафика, разрабатываемая в ИСП РАН, состоит из обрабатывающих модулей. В рамках данной задачи использовалось несколько схем обрабатывающих модулей, которые объединяются в схемы обработки. На схемах ниже оранжевым обозначены модули, которые уже были реализованы в системе, а фиолетовым - те, которые появились в процессе работы над задачей.

Сначала использовались схемы без сборки TCP-сессий: для них использовался оригинальный алгоритм LASER, который работает с первыми N пакетами, а не с первыми n-байт полезной нагрузки, так как конечной сигнатурой LASER является наибольшая сигнатура-пакета, а не потока.

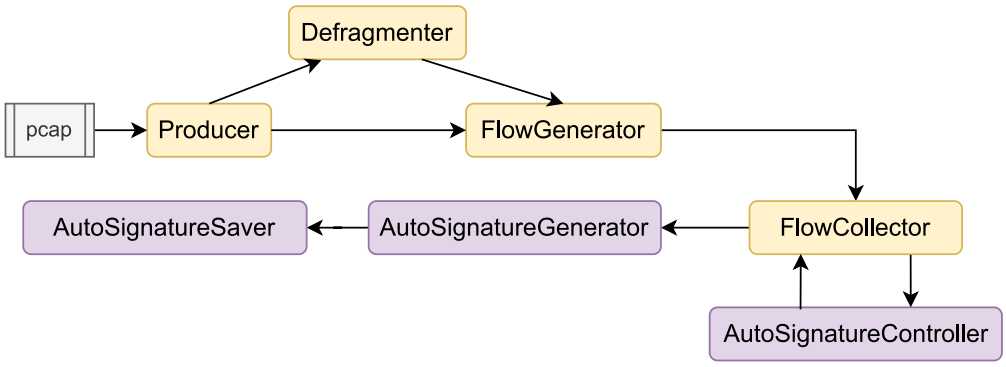

Рис. 6: Схема генерации сигнатуры без сборки TCP-сессий.

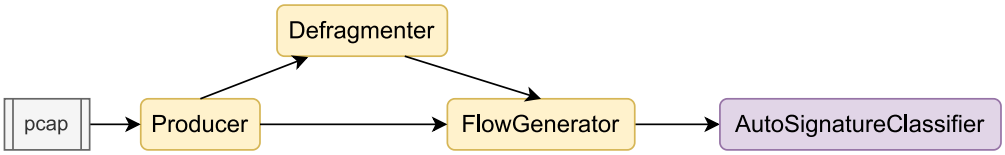

Рис. 7: Схема классификации трафика без сборки TCP-сессий.

Поговорим про каждый модуль более подробно:

- **Producer**: обрабатывает поступающий ему на вход pcap-файл, разделяя его на пакеты. Также может поставлять пакеты с интерфейса устройства в режиме реального времени.
- **Defragmenter**: осуществляет дефрагментацию IP пакета, если он был разделён на фрагменты.
- **FlowGenerator**: приписывает каждому пакету номер потока, к которому он принадлежит по IP-адресу источника, IP-адресу назначения, порту источника, порту назначения и используемый протокол транспортного уровня. Есть возможность объединять потоки, идущие в две стороны.
- **FlowCollector**: сохраняет приходящие пакеты и отправляет их в модуль AutoSignatureController и ждёт от него сигнала. Как только приходит сигнал от AutoSignatureController с идентификатором потока, FlowCollector отправляет все сохранённые пакеты этого потока в модуль AutoSignatureGenerator.
- **AutoSignatureController**: собирает статистику по каждому потоку и отправляет сигнал FlowCollector с идентификатором того потока, который выполнил некоторые требования выбранного алгоритма. Например, для LASER это ограничение по пакетам: в потоке должно присутствовать хотя бы N пакетов, последующие пакеты будут проигнорированы.
- **AutoSignatureGenerator**: выполняет генерацию сигнатуры на основе приходящих пакетов и передаёт полученную сигнатуру в AutoSignatureSaver.
- **AutoSignatureSaver**: выполняет десериализацию сигнатуры и сохраняет в файл в формате .json.
- **AutoSignatureClassifier**: классифицирует приходящий трафик, сопоставляя каждый пакет с набором сигнатур. Здесь считается, что сигнатура это сигнатура-пакета.

Для всех последующих других методов и других модификаций алгоритма LASER использовалась схема со сборкой TCP-сессии.

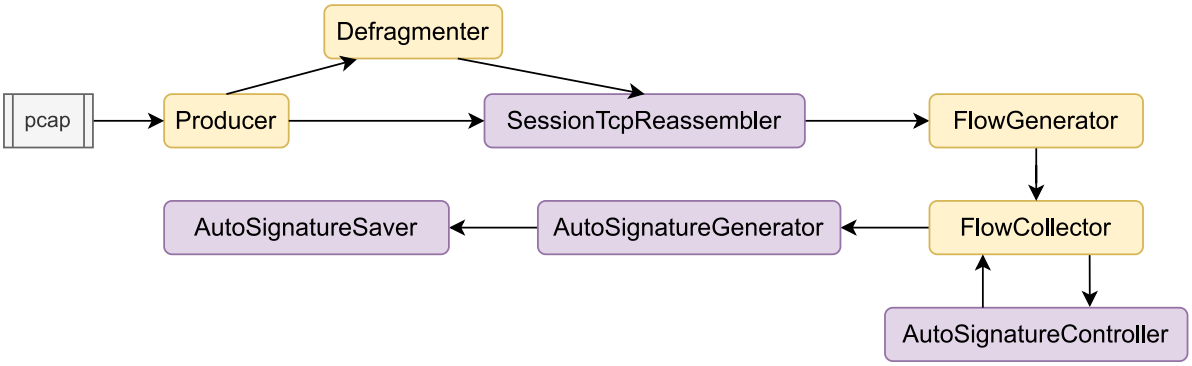

Рис. 8: Схема генерации сигнатуры со сборкой TCP-сессий.

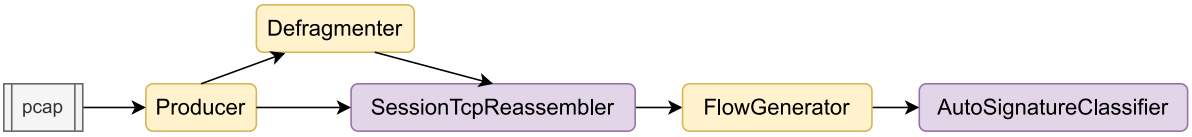

Рис. 9: Схема классификации трафика со сборкой TCP-сессий.

Отличия от предыдущих схем:

- После дефрагментации пакетов стоит модуль SessionTcpReassembler. Он собирает TCP-сессию и отправляет полезную нагрузку дальше после завершения сборки (или при срабатывании настраиваемого таймера), все последующие модули работают с этой полезной нагрузкой. Если пришёл UDP-пакет, то просто пропускается дальше.
- AutoSignatureController использует ограничение на необходимую длину полезной нагрузки.
- AutoSignatureClassifier использует уже сигнатуры-потоков, так как сопоставление происходит по полезной нагрузке потока.

В ходе данной работы были разработаны следующие модули:

1. AutoSignatureGenerator,
2. AutoSignatureClassifier,
3. AutoSignatureController,
4. AutoSignatureSaver,
5. SessionTcpReassembler.

Так как все модули системы были реализованы на языке программирования `C++`, то новые модули были реализованы также на нём. `C++` позволяет писать высокопроизводительные программы, что является необходимостью для обработки высокоскоростного трафика в режиме реального времени.

Также для совместимости с SessionTcpReassembler были модифицированы модули FlowGenerator и FlowCollector, которые теперь позволяют работать не только с пакетами в отдельности, но и с полезной нагрузкой TCP-сессии.

В конечном итоге после внедрения модулей было получено 2 пайплайна. Первый позволяет генерировать сигнатуры потоков на основе их полезной нагрузки, а второй использует эти сигнатуры для классификации трафика. Разделение на 2 пайплайна выполнено из соображения их независимости.

6 Заключение

В данной работе был разработан и реализован автоматический генератор сигнатур на основе полезной нагрузки сетевого трафика и классификатор трафика, основанный на сопоставлении этих сигнатур, в соответствии с использующимся протоколом или приложением в режиме реального времени.

Были выполнены следующие задачи:

1. Проведено исследование литературы по соответствующей теме.
2. Был собран набор сетевых трасс для последующего тестирования и сравнения методов.
3. Был выбран оптимальный метод для автоматической генерации сигнатур.
4. Был выбран оптимальный набор параметров метода для каждого тестируемого протокола и приложения.
5. Были рассмотрены ограничения выбранного метода.
6. Были реализованы автоматический генератор сигнатур и классификатор как модули в инструменте анализа высокоскоростного сетевого трафика, разрабатываемый в ИСП РАН.

Список литературы

[1] Internet Assigned Numbers Authority. — https://www.iana.org, дата обращения 05.02.2024.

[2] Dusi Maurizio, Crotti Manuel, Gringoli Francesco, Salgarelli Luca. Tunnel hunter: Detecting application-layer tunnels with statistical fingerprinting // Computer Networks. — 2009. — Vol. 53, no. 1. — P. 81–97.

[3] Гетьман А. И., Евстропов Е. Ф., Маркин Ю. В. Анализ сетевого трафика в режиме реального времени: обзор прикладных задач, подходов и решений. // Препринт ИСП РАН. — 2015. — Т. 28. — С. 1–52.

[4] Erman Jeffrey, Mahanti Anirban, Arlitt Martin. Qrp05-4: Internet traffic identification using machine learning // IEEE Globecom 2006 / IEEE. — 2006. — P. 1–6.

[5] Singh Sumeet, Estan Cristian, Varghese George, Savage Stefan. Automated Worm Fingerprinting. // OSDI. — Vol. 4. — 2004. — P. 4–4.

[6] Kim Hyang-Ah, Karp Brad. Autograph: Toward Automated, Distributed Worm Signature Detection. // USENIX security symposium / San Diego, CA. — Vol. 286. — 2004.

[7] Newsome James, Karp Brad, Song Dawn. Polygraph: Automatically generating signatures for polymorphic worms // 2005 IEEE Symposium on Security and Privacy (S&P'05) / IEEE. — 2005. — P. 226–241.

[8] Перспективный мониторинг. — https://amonitoring.ru/service/snort/, дата обращения 05.02.2024.

[9] Park Byung-Chul, Won Young J, Kim Myung-Sup, Hong James W. Towards automated application signature generation for traffic identification // NOMS 2008-2008 IEEE Network Operations and Management Symposium / IEEE. — 2008. — P. 160–167.

[10] Szabó Géza, Turányi Zoltán, Toka László, Molnár Sándor and. Automatic protocol signature generation framework for deep packet inspection // 5th International ICST Conference on Performance Evaluation Methodologies and Tools. — 2012.

[11] Wang Yu, Xiang Yang, Zhou Wanlei, Yu Shunzheng. Generating regular expression signatures for network traffic classification in trusted network management // Journal of Network and Computer Applications. — 2012. — Vol. 35, no. 3. — P. 992–1000.

[12] Vinoth George C, Ewards V. Efficient regular expression signature generation for network traffic classification // International Journal of Science and Research (IJSR). — 2013.

[13] Snort. — https://www.snort.org, дата обращения 05.02.2024.

[14] Ye Mingjiang, Xu Ke, Wu Jianping, Po Hu. Autosig-automatically generating signatures for applications // 2009 Ninth IEEE International Conference on Computer and Information Technology / IEEE. — Vol. 2. — 2009. — P. 104–109.

[15] Santos Alysson Feitoza. Automatic Signature Generation : Ph.D. thesis / Alysson Feitoza Santos ; Federal University of Pernambuco. — 2009. — https://www.cin.ufpe.br/~tg/2009-1/afs5.pdf.

[16] Goo Young-Hoon, Shim Kyu-Seok, Lee Su-Kang, Kim Myung-Sup. Payload signature structure for accurate application traffic classification // 2016 18th Asia-Pacific Network Operations and Management Symposium (APNOMS) / IEEE. — 2016. — P. 1–4.

[17] Shim Kyu-Seok, Goo Young-Hoon, Lee Dongcheul, Kim Myung-Sup. Automatic Payload Signature Update System for the Classification of Dynamically Changing Internet Applications // KSII Transactions on Internet and Information Systems (TIIS). — 2019. — Vol. 13, no. 3. — P. 1284–1297.

[18] Ning Kang, Ng Hoong Kee, Leong Hon Wai. Finding patterns in biological sequences by longest common subsequencesand shortest common supersequences // Sixth IEEE Symposium on BioInformatics and BioEngineering (BIBE'06) / IEEE. — 2006. — P. 53–60.

[19] Shim Kyu-Seok, Yoon Sung-Ho, Lee Su-Kang, Kim Myung-Sup. SigBox: Automatic Signature Generation Method for Fine-Grained Traffic Identification. // Journal of Information Science & Engineering. — 2017. — Vol. 33, no. 2.

[20] Karp Richard M, Rabin Michael O. Efficient randomized pattern-matching algorithms // IBM journal of research and development. — 1987. — Vol. 31, no. 2. — P. 249–260.

[21] Boyer Robert S, Moore J Strother. A fast string searching algorithm // Communications of the ACM. — 1977. — Vol. 20, no. 10. — P. 762–772.

[22] Xie Linquan, Liu Xiaoming, Yue Guangxue. Improved pattern matching algorithm of BMHS // 2010 Third International Symposium on Information Science and Engineering / IEEE. — 2010. — P. 616–619.

[23] Zhou Yansen, Pang Ruixuan. Research of Pattern Matching Algorithm Based on KMP and BMHS2 // 2019 IEEE 5th International Conference on Computer and Communications (ICCC) / IEEE. — 2019. — P. 193–197.

[24] Bittorent. — `https://www.bittorent.com`, дата обращения 05.02.2024.

[25] Wireshark. — `https://www.wireshark.org`, дата обращения 05.02.2024.