Министерство образования и науки Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего профессионального образования
«Уральский федеральный университет
имени первого Президента России Б. Н. Ельцина»

Институт математики и компьютерных наук
Кафедра вычислительной математики

Поиск вредоносной активности в DNS трафике

Выпускная квалификационная работа
на степень бакалавра по направлению
02.03.02 Фундаментальная информатика
и информационные технологии
студента группы МК-420002
Меньших Ивана Александровича

Научный руководитель
доцент Солодушкин Святослав Игоревич

Допустить к защите:

_____

«____» _____ 2016 г.

Екатеринбург
2016

РЕФЕРАТ

Выпускная квалификационная работа на степень бакалавра содержит 47 страниц, 7 рисунков, 7 таблиц, 9 источников, 1 приложение.

Ключевые слова: ВРЕДОНОСНАЯ АКТИВНОСТЬ, DNS ТРАФИК, ДОМЕН, БОТНЕТ, ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ, ЗАРАЖЕННЫЕ ХОСТЫ, C&C СЕРВЕР

Объект исследования - логи DNS сервера.

Цель работы - защита корпоративной сети от вредоносных коммуникаций с C&C серверами, фильтрация трафика.

Результат работы - программный продукт для нахождения вредоносных доменов в логах DNS сервера а также для выявления паттернов взаимодействия групп зараженных хостов с зараженными доменами.

Содержание

1 Введение в проблему поиска вредоносной активности

Важнейшей задачей, стоящей перед системными администраторами, является защита корпоративной сети и пресечение любых вредоносных взаимодействий клиентов сети и сети интернет. Схожая задача стоит перед сервисами, предоставляющими услуги DNS фильтрации. Анализ логов DNS сервера позволяет выделить в общей массе запросов «подозрительные» (то есть те, которые соответствуют коммуникации зараженного хоста и, например, управляющего C&C сервера).

В реальности, анализ DNS логов проводить крайне трудно, потому что каждый клиент генерирует большое количество запросов. Даже при заходе на сайт пользователь генерирует множество запросов. Как правило, при загрузке страницы происходят обращения на сторонние сервера для загрузки стилей, скриптов, изображений и так далее. Даже когда пользователь ничего не делает, его ПК генерирует запросы. Это могут быть обновления программного обеспечения, синхронизация часов и так далее. Особенно это заметно в новых версиях Windows: операционная система старается «радовать» пользователя своей интерактивностью и постоянно запрашивает новости, курсы валют, отправляет различную информацию о его действиях на свои сервера для «персонализации» взаимодействия.

В работе будут рассмотрены различные способы анализа DNS трафика, на базе которых можно выделить домены, которые, вероятно, используются для коммуникации между зараженным хостом и ботмастером.

Общая идея заключается в анализе паттернов взаимодействия зараженных клиентов и C&C серверов. Кроме того, речь будет идти о фильтрации запросов, что является важным аспектом, ведь известно, что <>. Также будут сделаны предположения о структуре паттернов взаимодействия и предложены методы анализа.

2 Формальная постановка задачи

Для того, чтобы приступить к решению этой проблемы, необходимо формализовать задачу.

Дано:

1. DNS логи ($querylog$). Определим их как множество Q. Каждая запись в нем представима в виде вектора признаков $\vec{query} \in Q$, который обязательно содержит $source\_ip$ (ip адрес клиента, который совершает запрос) и $domain$ (домен, который пользователь желает «разрешить»). Кроме того, в нем могут содержаться дополнительные данные, например время, когда был совершен запрос. Подробно структура $\vec{query}$ будет описана далее.

2. Белый список доменов ($whitelist$). Будем использовать «самые часто-посещаемые» домены, для этого воспользуемся списками от Alexa и Quantcast)

3. Черный список доменов ($blacklist$). Будем использовать базы компании SkyDNS и прочие источники.

Необходимо:

1. Выбрать из $querylog$ подмножество запросов $\{\vec{query}_1, \vec{query}_2, \vec{query}_3, \dots\}$, которые были совершены к вредоносным доменам и выделить имена этих доменов. На самом деле наибольший интерес представляют сами доменные имена, так как по ним легко осуществлять блокировку, что и является конечной целью.

2. Выделить общие паттерны взаимодействия клиентов и вредоносных доменов, описать внешний вид графа запросов.

2.1 Структура $\vec{query}, whitelist, blacklist$

Структура *query* приводится в таблице 2.1. В ней описаны типы исходных данных о запросах, которые будут использованы в работе. Стоит обратить внимания, что в «обычных условиях» получить их непросто (а именно, если использовать какую либо стандартную реализацию DNS сервера). Процесс сбора этих данных будет описан в главе 7.

Таблица 2.1 – Описание структуры $\vec{query}$

Поле	Описание
domain	Домен, который был запрошен пользователем
source_ip	IP адрес пользователя
rcode	Код возврата DNS сервера
qtype	Код запрашиваемого типа записи
timestamp	Временная метка получения сервером запроса

Структура *whitelist* и *blacklist* представлена в таблице 2.2. Это просто списки доменных имен с их категориями. В рамках работы, нас будет интересовать только наличие категории «вредоносный домен». Это единственная априорная информация, которая потребуется для осуществления поиска вредоносных доменов.

Таблица 2.2 – Описание структуры $whitelist, blacklist$

Поле	Описание
domain	Доменное имя
cats	Список категорий, к которым принадлежит домен

2.2 Словарь

В таблице 2.3 приведены основные понятия (сокращения), которые будут использованы в данной работе.

Таблица 2.3 – Основные понятия, используемые в работе

Термин	Описание
Домен	Символьное имя, служащее для идентификации областей — единиц административной автономии в сети Интернет — в составе вышестоящей по иерархии такой области.
TLD	Домен верхнего (первого) уровня (например .com, .eu)
iLD	Домен i уровня
Хост	Клиентский ПК
Ботнет	Сеть из зараженных хостов, управляемая ботмастером через C&C сервер
C&C	Управляющий сервер ботнета
whois	Сервис, позволяющий узнать информацию о регистрации доменного имени, а также о его владельце
pDNS	Техника, позволяющая отслеживать изменения в изменении соответствия доменных имен ip адресам
Bind	Одна из реализаций DNS сервера

3 Ботнеты

3.1 Описание

Ботнет - сеть инфицированных хостов, управляемая извне. Бот, по своей сути, это вредоносное программное обеспечение, которое позволяет злоумышленнику удаленно управлять устройством пользователя.

Заражение подобным программным обеспечением может происходить через уязвимости в используемом ПО (например, через браузер и эксплуатации 0-day exploit, который позволяет исполнить произвольный код за пределами «песочницы» браузера). Кроме того, заражение может произойти с помощью «социальной инженерии», где вредоносное ПО маскируется под полезное содержимое, например, вложение в электронном письме в виде договора, счета или иной важной информации.

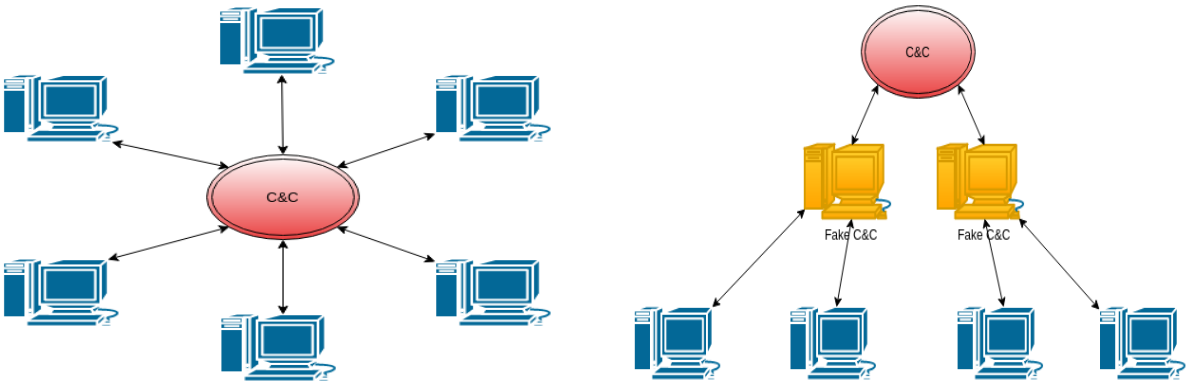
Проблема детектирования ботнетов стоит очень остро, так как они растут вместе с интернетом. Например, один из самых больших - ботнет Ponmocup, по данным компании FoxIT, достигал размера 15000000 хостов.

Очень важно научиться выявлять ботнеты, а именно, в рамках данной работы, нас интересуют доменные имена, которые используют C&C сервера. Это поможет нам предупредить пользователей, чьи ПК проявляют «аномальную» активность, схожую с активностью члена ботнета.

3.2 Топология построения ботнетов

Можно выделить две основных топологии построения ботнетов, которые подробно рассмотрены в работе [1].

Топология, изображенная на рис. 3.1а является самой часто встречающейся и простой. Зараженные хосты напрямую обращаются к C&C серверу. Основная ее уязвимость - централизация. Благодаря этому аспекту, отследить управляющий сервер гораздо проще. Соответственно, и обезвредить такой ботнет легче. Из плюсов стоит отметить быструю коммуникацию, так как отсутствует «прослойка» между клиентом и сервером.



(a) «Звезда» (b) «Дерево»

Рисунок 3.1 – Популярные варианты топологии ботнета

Как расширение данной топологии, можно рассматривать вариант, в котором используется несколько серверов. Это убирает проблему с единой точкой отказа, а также позволяет балансировать нагрузку по географической принадлежности (для общения с хостом используется сервер, который к нему физически ближе). Минус - дополнительные затраты на сервера, а также сложность в синхронизации между серверами.

Топология ботнета (рис. 3.1b) встречается реже, но является более устойчивой. Смысл заключается в том, что зараженные хосты общаются с управляющим сервером через «прокси», в качестве которого выступают другие зараженные хосты. Это позволяет скрыть реальный C&C сервер, что препятствует его обнаружению. Кроме того, ботнет проще будет «продать» по частям, продавая контроль над некоторым поддеревом данного графа (а дерево может быть достаточно глубоким).

Также встречается топология вида P2P, где хосты соединены между собой и образуют небольшие плотные графы. Это самый устойчивый вариант, но им сложно управлять. В нем просто нет места для C&C сервера и команды, фактически, передаются какому-либо хосту, который распространяет эту команду всем остальным.

3.3 Стратегии уклонения от обнаружения

Для того, чтобы предотвратить обнаружение управляющий серверов и уничтожение ботнета, ботмастеры используют различные стратегии, препятствующие этому [1].

3.3.1 IP Flux (fast flux)

Один из вариантов - использовать много ip адресов для одного доменного имени и маленький TTL для A записи в DNS пакете. Данный прием можно усложнить, меняя также ip адреса соответствующий NS серверов.

Изначально, данный прием позволял легитимными сайтами балансировать нагрузку между серверами еще на этапе разрешения доменного имени. Этот способ до сих пор используется разными сайтами (например google.com).

3.3.2 Domain Flux

Другой вариант - использовать много доменных имен, разрешающихся в один ip адрес. Именно этот подход чаще всего используется в настоящее время. Это связано с исчерпанием IPv4 диапазона и, в то же время, доменное имя практически ничего не стоит. Кроме того, владея 2LD доменом, можно легко генерировать множество 3LD доменов (example.com -> *.example.com). Как правило, для этого используются специальные алгоритмы, которые генерируют домены на основе системного времени. Класс таких доменов называется DGA. Они являются короткоживущими (несколько часов - несколько недель) и используются для одноразовой коммуникации.

3.4 Коммуникация

Процесс коммуникации, приведенный на рисунке 3.2 можно разделить на несколько шагов.

Описание схемы (рис 3.2):

1. Зараженный хост и управляющий сервер синхронизируют время путем отправки запроса по протоколу NTP к некоторому серверу точного времени. Эта стадия происходит не всегда, если не требуется синхронизации по минутам/секундам. Зачастую, хватает точности до суток.

2. Хост и C&C сервер генерируют доменные имена, основываясь на текущей временной метке.

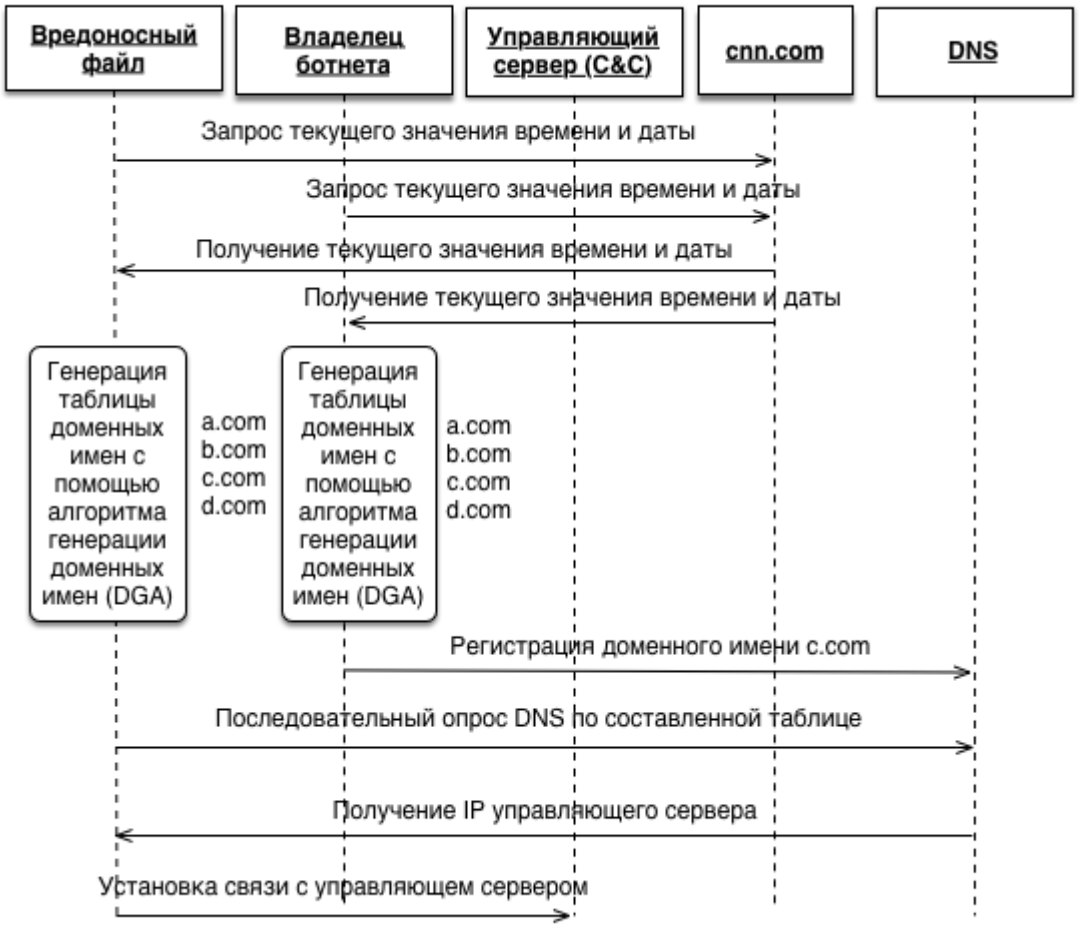


Рисунок 3.2 – Установка связи между инфицированным хостом и C&C сервером

3. C&C сервер регистрирует эти доменные имена.

4. Клиент опрашивает сгенерированные доменные имена по списку и пытается получить ip адрес.

5. Клиент устанавливает соединение с C&C сервером и получает необходимые команды/обновления.

Таким образом, теперь мы имеем базовое представление о ботнетах, их структуре и поведении.

4 Групповая активность

4.1 Основные предположения об активности ботнетов

Один из подходов поиска ботнетов - отслеживание групповой активности клиент-домен. Основываясь на работе [2], сделаем несколько предположений:

1. Зараженных хостов в сети фиксированное количество.

2. Взаимодействие зараженных хостов и C&C сервера проявляется периодически.

3. Как правило, используется техника domain flux, которая была описана в пункте 3.3.2 (используется много доменных имен и мало ip адресов).

4. В случае аномального взаимодействия хостов и домена можно выделить «периодическое» взаимодействие узкого круга пользователей с конкретным доменом.

Исходя из данных предположений, можно составить алгоритм, который выделит подозрительные коммуникации между группами хостов и доменом.

4.2 Алгоритм обнаружения групповой активности

Алгоритм поиска групповой активности можно разделить на 3 стадии: сбор информации о запросах пользователей, фильтрация запросов и детектирование подозрительной активности.

Изначально, мы делим все *querylog* на «окна», в данном случае использовалось окно длины 1 час. Выбрано это число из таких соображений: частая коммуникация не свойственна ботнетам, поэтому не имеет смысл использовать окно меньшего размера. Кроме того, при слишком маленьком окне, высок шанс ложного срабатывания алгоритма, а при слишком большом - слабой чувствительности.

Первая функция группирует все пары запросов ($client\_ip$, $domain$) по домену. Подобная агрегация позволяет получить представление о том, кто именно запрашивает конкретный домен.

function SEARCHINFO(*querylog* [by hour])
    data ← *Dictionary*
    for (*client_ip*, *domain*) in *querylog* **do**
        data[*domain*].add(*client*)
    end for
    return *data*
end function

Следующая функция фильтрует полученные данные. Нам необходимо исключить все доменные имена, которые есть в *whitelist*, а также те доменные имена, которые запрашиваются малым количеством пользователей (хорошим вариантом будет значение $threshould \in 3, 4, 5, 6, 7$)

function FILTERQUERIES(*data*, *whitelist*, *threshould*)
    for *domain* in *data* **do**
        if *domain* in *whitelist* OR length(data[domain]) < threshould **then**
            ip_list ← *data*[*domain*]
            data.remove(*domain*)
            data.pop_values(*ip_list*)
        end if
    end for
    return *data*
end function

Последняя функция выполняет предварительную обработку *querylogs*, выбирает доменные имена, к которым в разные временные промежутки обращаются, в основном, одни и те же пользователи, что, в рамках наших начальных предположений, говорит о вредоносности такого домена

function DETECTBOTNETDOMAIN(*querylogs*, *sim_score*)
    data_arr ← *List*
    botnet_domains ← *Set*
    for *querylog* in *querylogs* **do**
        curr_data ← *SearchInfo*(*querylog*)
        curr_data ← *FilterQueries*(*curr_data*)
        data_arr.append(*curr_data*)
    end for
    for $data_1$ in *data_arr* **do**

```
        for data_2 in data_arr do
            if data_1 ≠ data_2 then
                for domain in data_1 ∩ data_2 do
                    curr_sim ← Similarity(data_1[domain], data_2[domain])
                    if curr_sim > sim_score then
                        botnet_domains.add(domain)
                    end if
                end for
            end if
        end for
    end for
    return botnet_domains
end function
```

Несколько слов о функции *similarity*, она выражает «схожесть» между клиентами, которые запрашивают один и тот же домен. В работе [2] предлагается для этих целей использовать такую функцию:

$$similarity(ipList_1, ipList_2) = \frac{1}{2} * (\frac{C}{A} + \frac{C}{B})(A \neq 0, B \neq 0) \tag{4.1}$$

где

$$A = |ipList_1|, B = |ipList_2|, C = |ipList_1 \cap ipList_2|$$

Эксперименты показали, что есть более подходящая функция - расстояние Джаккара:

$$similarity(iplist_1, iplist_2) = \frac{|ipList_1 \cap ipList_2|}{|ipList_1 \cup ipList_2|} \tag{4.2}$$

Более того, интерпретация расстояния Джаккара интуитивно ясна и, в отличии от функции из работы [2], на практике работает ощутимо лучше.

5 Графовая модель запросов

5.1 Описание модели запросов

В главе 4 была рассмотрена активность групп пользователей относительно доменных имен. В данной главе «усовершенствуем» эту идею и перейдем к графовой модели запросов пользователей.

Рассмотрим двудольный неориентированный граф ($H \times D$, E), где H - множество хостов, D - множество доменных имён, (h_i, d_j)$\in E$, если пользователь h_i запрашивал домен d_j. На основе этой модели предлагается вычислять для каждого домена некоторую оценку, которая будет говорить о его вредоносности или же наоборот, о том, что домен «чист». Как мы увидим далее, такая же оценка вычисляется и для хостов, но ее в работе мы использовать не будем.

Идея заключается в том, чтобы передавать через «соседей» в графе оценку от доменов к клиентам и обратно. Этот процесс является итерационным.

Вспомним, что у нас есть *blacklist* и *whitelist*, в данном разделе будем пользоваться ими для инициализации оценок

5.2 Первый вариант ранжирования доменов

В работе [3] предлагается для оценки использовать процесс случайного блуждания по графу. Использование «блуждания» обусловлено тем, что граф получается слишком большой, поэтому они, фактически, вместо точной оценки вычисляют ее приближение. В этой работе, мы будем рассматривать граф, который генерируется на основе 3 часов логов. Так как он помещается в RAM, то мы не будем организовывать случайное блуждание. Мы будем «честно» на каждой итерации проходить от каждой вершины ко всем ее соседям и пересчитывать оценку.

Обозначим правила пересчета. На каждой итерации, они симметрично вычисляют оценку для каждой вершины из доли вначале для хостов, потом

для доменов по формулам 5.1, 5.2:

$$black\_score(h_i) = \sum_{d_j:(h_i,d_j)\in E} \frac{black\_score(d_j)}{deg(d_j)} \tag{5.1}$$

$$white\_score(h_i) = \sum_{d_j:(h_i,d_j)\in E} \frac{white\_score(d_j)}{deg(d_j)} \tag{5.2}$$

Возникает закономерный вопрос, а зачем нам так много различных оценок, нельзя ли ограничиться просто *black_score*? Как оказалось, его недостаточно, потому что любой популярный «чистый» домен может получить высокую оценку по этому параметру просто из за своей популярности (в графе к нему будет идти много ребер, как от зараженных, так и от «чистых» хостов). Именно для этого вводится *white_score*, который у популярных доменов будет также высоким.

Теперь нам нужно «взвесить» две полученных оценки, что мы сделаем за счет *union_score* по формуле 5.3:

$$union\_score(d_i) = \frac{black\_score(d_i)}{black\_score(d_i) + white\_score(d_i)} \tag{5.3}$$

По сущетсву, *union_score* это отношение «негативных» оценок ко всем остальным.

5.2.1 Алгоритм вычисления *union_score*

Опишем алгоритм пересчета всех оценок на двудольном графе

function CalcScores($graph, max\_iter, whitelist, blacklist, init\_weight$)
    $black\_score = Dictionary$
    $white\_score = Dictionary$
    $union\_score = Dictionary$
    for *domain* in *blacklist* **do**
        $black\_score[domain] = init\_weight$
    end for
    for *domain* in *whitelist* **do**
        $white\_score[domain] = init\_weight$
    end for

while $max\_iter > 0$ **do**
for $h_i \in H$ **do**
$black\_score(h_i) = \sum_{d_j:(h_i,d_j)\in E} \frac{black\_score(d_j)}{deg(d_j)}$
$white\_score(h_i) = \sum_{d_j:(h_i,d_j)\in E} \frac{white\_score(d_j)}{deg(d_j)}$
end for
for $d_j \in D$ **do**
$black\_score(d_j) = \sum_{h_i:(h_i,d_j)\in E} \frac{black\_score(h_i)}{deg(h_i)}$
$white\_score(d_j) = \sum_{h_i:(h_i,d_j)\in E} \frac{white\_score(h_i)}{deg(h_i)}$
end for
$max\_iter \leftarrow max\_iter - 1$
end while
for $d_i \in D$ **do**
$union\_score(d_i) = \frac{black\_score(d_i)}{black\_score(d_i)+white\_score(d_i)}$
end for
return $union\_score$
end function

5.3 Второй вариант ранжирования доменов

Теперь мы используем ту же графовую модель, но с другим правилом пересчета. Это правило упоминается в работе [4]. Само правило выглядит следующим образом (формула 5.4):

$$rank\_score(d_j) = \sum_{h_i:(h_i,d_j)\in E} \frac{rank\_score(h_i)}{deg'(h_i)} \tag{5.4}$$

Эта формула схожа с 5.1, но есть два серьезных отличия.

Во-первых, вместо $deg(h_i)$ вычисляем $deg'(h_i)$, который определен не как степень вершины в текущем графе, а как средняя степень текущей вершины по всем графам за некоторый промежуток времени (например за неделю).

Во-вторых, на этапе инициализации рангов используются оценки разных знаков (положительные для «чистых» и отрицательные для «вредоносных»), в отличии от алгоритма 5.2.1, где при инициализации используются только положительные оценки.

Этот алгоритм очень напоминает алгоритм $PageRank$, но, на самом деле, между ними много отличий, которые приведены в таблице 5.1.

Таблица 5.1 – Сравнение $rank\_score$ и $PageRank$

$rank\_score$	$PageRank$
Редкое посещение домена «здоровыми» клиентами увеличивает шанс того, что домен вредоносный	Больше связей с хорошими страницами увеличивает шанс того, что страница получит высокий ранг
Два типа вершин	Один тип вершин
Неориентированный граф	Ориентированный граф
Фактически, мы имеет разбиение на 3 класса: вредоносные домены, чистые домены и неизвестные домены	Нет разбиения, только ранжирование
Вычисляем риск	Вычисляем популярность
Очень сильно зависит от начальной инициализации весов	Не зависит от начальных весов

Это правило пересчёта также похоже на алгоритм $HITS$, если мы превратим наш граф в ориентированный (все рёбра от хостов к доменам) и будем использовать $auth(d_i)$.

5.3.1 Алгоритм вычисления $rank\_score$

function CalcRank($graph, max\_iter, whitelist, blacklist, init\_weight, deg'$)
    $rank\_score = Dictionary$
    for $domain$ in $blacklist$ **do**
        $rank\_score[domain] = -init\_weight$
    end for
    for $domain$ in $whitelist$ **do**
        $rank\_score[domain] = init\_weight$
    end for
    while $max\_iter > 0$ **do**

for $h_i \in H$ **do**

$rank\_score(h_i) = \sum_{d_j:(h_i,d_j)\in E} \frac{rank\_score(d_j)}{deg'(d_j)}$

end for

for $d_j \in D$ **do**

$rank\_score(d_j) = \sum_{h_i:(h_i,d_j)\in E} \frac{rank\_score(h_i)}{deg'(h_i)}$

end for

$max\_iter \leftarrow max\_iter - 1$

end while

return $rank\_score$

end function

5.4 Структура сообществ в графе

Усложним нашу первоначальную модель графа. У нас вновь будет двудольный неориентированный граф ($H \times D$, E), где H - множество хостов, D - множество доменных имён. Вся разница в правиле построения рёбер. Ребро (h_i, d_j, w_k) $\in E$, если пользователь h_i запрашивал домен d_j и этот запрос завершился ошибкой (значение поля *rcode* из DNS-пакета > 0 [6], оно присутствует в *querylog*), вес w_k вычисляется как количество запросов, завершившихся ошибкой, от h_i к d_j.

Впервые такой граф фигурировал в работе [5]. Пример изображен на рисунке 5.1.

Данный рисунок был отфильтрован от компонент связности, состоящих из двух вершин. Таких компонент в нем более чем 99% от общего числа компонент связности и для нас они не представляют никакого интереса, так как из за их малого размера мы не может сделать достоверный вывод о их «природе». Кроме того, был взят именно произвольный подграф из за большого размера исходного графа, на котором нельзя визуально увидеть структуру из за обилия ребер.

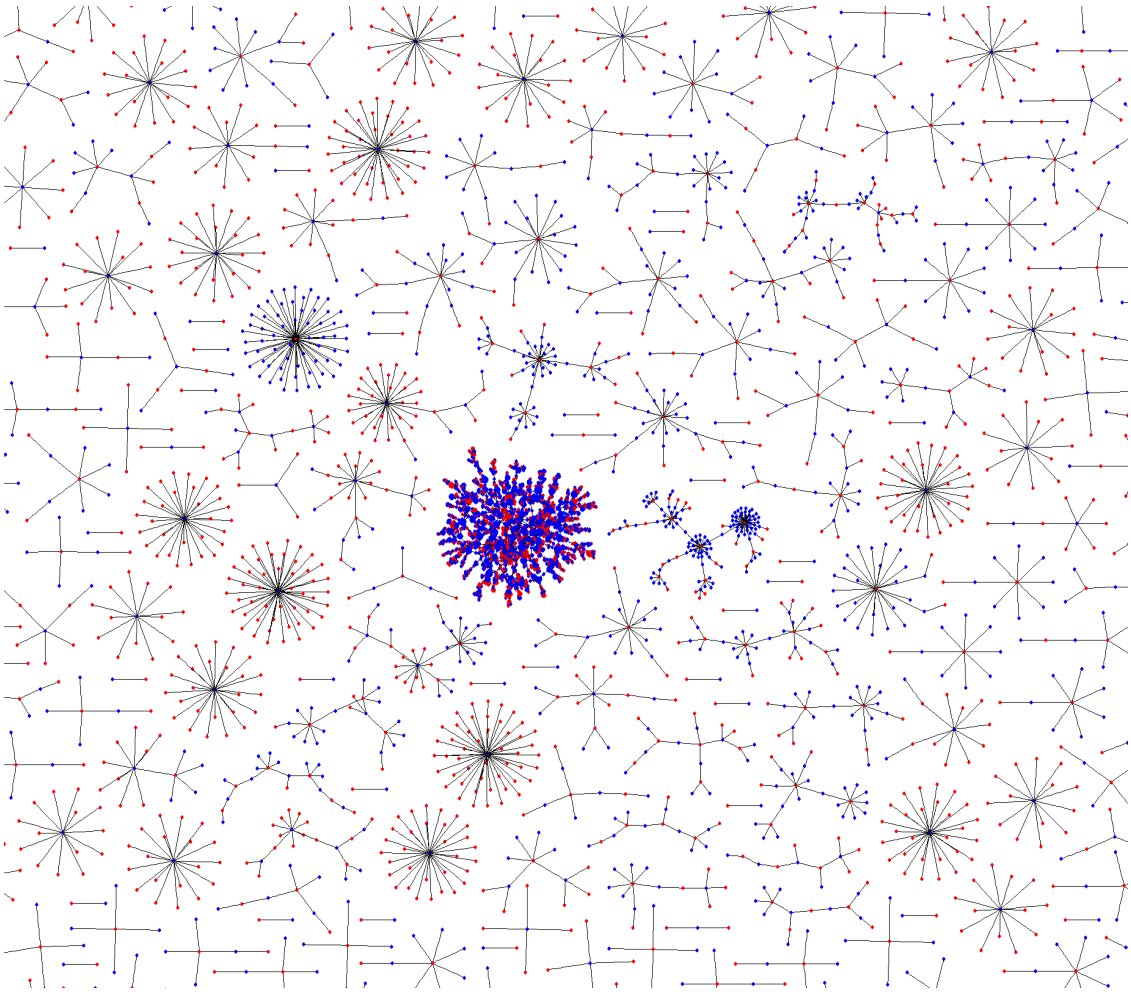


Рисунок 5.1 – Граф неудачных DNS-запросов (синие - хосты, красные - домены)

Из него можно выделить типичные паттерны подграфов.

1. «Звезда» с доменом в центре. Причины появления:
   a) домен, который существовал ранее, но больше не существует;
   b) домен, который является кандидатом на C&C ботнета;
   c) типичная ошибка настройки клиента (например префикс DNS);
   d) другое.
2. «Звезда» с хостом в центре. Причины появления:
   a) ошибка настройки клиента;
   b) перебор возможных доменных имен C&C сервера;
   c) другое.
3. Ребро из одного хоста к одному домену. Соответствует запросу редкого домена.

4. Плотный двудольный подграф, соответствует «сильной» связи между группами пользователей и группой доменов. Может быть порожден группой клиентов - членов одного и того же ботнета.

5. «Дерево», которое представимо в виде объединения компонент, перечисленных выше

5.4.1 Алгоритм декомпозиции графа

Общая идея состоит в построении таких графов и выделении из них плотных подграфов, для которых строится множество признаков, по которому строится классификатор. Опишем основные процедуры этого процесса

В первую очередь, нам нужно построить граф ($H \times D$, E), при этом выбросим «мусор». Для этого, при построении графа, мы удалим вершины, соответствующие доменам из *whitelist* (предполагаю, что он содержит популярные домены, на подобие googe.com), также, мы удалим те рёбра, которые имеют $rcode \in (format\_error, serv\_failt, ...)$, т.е. ошибками на стороне DNS сервера, или же при формировании пакета [6].

function ConstructGraph(*querylog*, *whitelist*)
    $graph \leftarrow ((\emptyset \times \emptyset), \emptyset)$
    for *query* in *querylog* **do**
        if $query[rcode] == 0 \vee query[domain] \in whitelist$ **then**
            CONTINUE
        end if
        if $query[rcode] \in (format\_error, serv\_failt, ...)$ **then**
            CONTINUE
        end if
        $graph.add\_node(query[client\_ip])$
        $graph.add\_node(query[dname])$
        $graph.add\_edge((query[client\_ip], query[dname]))$
    end for
    return *graph*
end function

Следующий шаг - декомпозиция полученного графа. Процесс будет про-

ходить в два этапа: на первом этапе делается BFS, и для полученных компонент связности проводится процесс ко-кластеризации. Достаточно «плотные» компоненты мы будем рассматривать на следующем шаге.

```
function DecomposeGraph(graph, min_density)
    dense_subgraphs ← Array()
    connected_subgraphs ← BFS(graph)
    for graph in connected_subgraphs do
        partition ← coclustering(graph)
        for g in partition do
            if density(g) > min_density then
                dense_subgraphs.append(g)
            end if
        end for
    end for
    return dense_subgraphs
end function
```

Определим функцию, которая будет говорить о «плотности» подграфа таким образом

$$density = \frac{|E|}{|H| * |D|}$$

Что касается функции *coclustering*, существуют несколько способов сделать это. В статье [7] предлагается решать задачу 3-факторизации матрицы исходного графа. Опишем формальную постановку задачи 3-факторизации.

Пусть $A \in \mathbb{R}_+^{m \times n}$ - матрица сопряженности двудольного графа, $m = |H|, n = |D|$, мы хотим ее разложить на произведение трех матриц

$$U \in \mathbb{R}_+^{m \times k}, S \in \mathbb{R}_+^{k \times l}, V \in \mathbb{R}_+^{n \times l}$$

$$A \approx USV^T$$

Оптимизационная задача выглядит так:

$$\min_{U \geq 0, S \geq 0, V \geq 0} ||A - USV^T||^2$$

При условиях, что U и V - ортогональные, $||.||$ - норма Фробениуса.

Смысл этого разложения заключается в том, что матрицы U и V от-

вечают за кластеры по строкам и по столбцам , а матрица S описывает «тесноту связи» между парами кластеров (то есть между группой хостов и группой доменов). Описание решения данной задачи можно найти в [7].

В результате, мы получаем множество двудольных плотных подграфов, из каждого извлекаем список признаков

1. Медиана по всем TTL доменов в подграфе. Этот признак может выделить группы короткоживущих доменов, что может свидетельствовать о fast-flux доменах, которые часто используют ботмастера

2. Отношение количества доменов в графе на количество хостов

3. Лексические признаки:

   a) расстояние между распределение 1-грамм и 2-грамм до равномерного распределения. Для этого есть симметричное расстояние Кульбаха-Лейбнера. Довольно часто доменные имена генерируются с помощью ГСЧ, который, по умолчанию, выдает равномерное распределение;

   b) редакторское расстояние между всеми доменами в подграфе. Опять таки, для доменов сгенерированных с помощью ГСЧ будет большим;

   c) количество пересекающихся 3-грамм с 3-граммами из *whitelist* и *blacklist*. Также, к *whitelist* можно добавить словари популярных языков.

Это достаточно небольшая группа признаков, но уже она даёт результат в распознавании вредоносных подграфов. Больше признаков можно получить, используя *whois* и $pDNS$.

На последнем шаге обучаем классификатор на полученных признаках (для этого, естественно, нужно собрать обучающую выборку, можно опираться на *whitelist* и *blacklist*). Видно, что этот подход сильно отличается от упоминаемых ранее, так как он базируется не на доменных именах, а на целых подграфах $(H' \times D', E')$. Зато именно этот метод позволяет посмотреть на DNS трафик с более «высокого» уровня абстракции и выделить паттерны взаимодействия группы хостов - группы доменов.

6 Объединение подходов по оценке доменных имён

В данной работе были рассмотрены несколько подходов детектирования вредоносной активности. В разделе 5.4 мы получили готовый результат, но он применим только к группам пользователей и доменов. Теперь, нам нужно объединить остальные подходы, которые занимаются оценкой доменов, перечисленные в главах 4 и 5.

6.1 Описание признаков

Хорошим вариантом является использование техник машинного обучения, опираясь на выходные данные алгоритмов как на признаки доменов. Опишем те признаки, которые будем использовать. В качестве набора входных данных мы рассматриваем *querylog* за 3 часа.

1–3. *Similarity* (формула (4.1)) для всех пар окон.

4–6. *Similarity* (формула (4.2)) для всех пар окон.

7–9. Мощность пересечения множеств клиентов по домену (из главы 4) для всех пар окон.

10. *Black_Score* (формула (5.1)) по 3 часам.

11. *White_Score* (формула (5.2)) по 3 часам.

12. *Union_Score* (формула (5.3)) по 3 часам.

13. *Rank_Score* (формула (5.4)) по 3 часам.

6.2 Постановка задачи классификации

Этот набор признаков будем использовать в дальнейшем. Далее сформулируем задачу классификации (на самом деле это задача отбора классификаторов из множества всех обученных классификаторов)

Дано:

$\mathcal{X} = (X_1, \ldots, X_k)$	входные вектора признаков признаков, $X_i \in \mathbb{R}^n$
$\mathcal{Y} = (y_1, \ldots, y_k)$	вектор правильных ответов (вредоносный ли домен), $y_i \in 0, 1$
$(\mathcal{X}, \mathcal{Y})$	обучающая выборка
$(\mathcal{X}', \mathcal{Y}')$	тестовая выборка
ROC_AUC	Функция площади под ROC-кривой
CLF	Множество классификаторов
$PARAM$	Множество параметров для классификатора

Найти:

Классификатор $clf \in CLF$ такой, что $ROC\_AUC(\mathcal{X}', \mathcal{Y}') \to \max$

Стоит пояснить, что такое ROC_AUC и вспомогательные понятия, необходимые для его вычисления.

ROC_curve - характеристика качества классификатора, зависимость доли верных положительных классификаций от доли ложных положительных классификаций при изменении порога решающего правила. Для построения этой кривой необходимо вычислять FPR и TPR по формулам (6.1) и (6.2):

$$FPR(clf, (X, Y)) = \frac{\sum_{(x,y)\in(X,Y)}[clf(x) = +1][y = -1]}{\sum_{(x,y)\in(X,Y)}[y = -1]}, \tag{6.1}$$

$$TPR(clf, (X, Y)) = \frac{\sum_{(x,y)\in(X,Y)}[clf(x) = +1][y = +1]}{\sum_{(x,y)\in(X,Y)}[y = +1]}, \tag{6.2}$$

где clf - решающее правило, X - множество векторов, Y множество правильных ответов.

ROC_curve проходит через точки (0, 0), соответствующей максимальному значению порога и (1, 1), соответствующей минимальному значению порога. ROC_AUC вычисляется как площадь под кривой ROC_curve.

В качестве множества классификаторов возьмём $RandomForest$ и $AdaBoost$,

т.к. ансамблевые модели достаточно мощные и «неприхотливые» в плане входных данных (не требуют, например, нормализации данных).

Для каждого из них будем подбирать основные параметры. В случае с *AdaBoost* будем подбирать количество деревьев и скорость обучения. В случае же с $RandomForest$ будем выбирать количество деревьев, информационный критерий и максимальное количество признаков, по которым можно производить разделение.

6.3 Алгоритм обучения классификатора

Перечислим шаги, необходимые для получения итогового классификатора

1. Весь объем *querylog* делим на части по 3 часа в каждой, каждая часть будет использоваться как входные данные для итогового алгоритма. Шаги далее будут описаны для работы с одной такой частью.

2. Вычисляем признаки 1 – 9 из 6.1. Для их вычисления не требуется никакой дополнительной информации

3. Строим двудольный граф $(H \times D, E)$.

4. Берем исходные *blacklist* и *whitelist*, перемешиваем и делаем 4 независимых разбиения на обучающую и тестовую выборки в соотношении 70/30 (такой подход именуется $KFoldCV$).

5. Для каждого такого разбиения (X_train, y_train), (X_test, y_test)

   a) вычисляем признаки 10–13 (используя для начальной инициализации (X_train, y_train);

   b) для каждой пары $(clf_i, param_i) \in CLF \times PARAM$

   - обучанем классификатор clf_i с параметрами $param_i$ на множестве (X_train, y_train);
   - вычисляем $ROC\_AUC(X\_test, y\_test)$.

6. Вычисляем

$$clf, param = \underset{(clf_i, param_i) \in CLF \times PARAM}{\arg\max} mean(ROC\_AUC(X\_test, y\_test)) \tag{6.3}$$

7. Берем целиком *whitelist* и *blacklist* и вычисляем признаки 10 – 13.

8. Обучаем классификатор clf с параметрами $param$ на всей выборке.

Итоговый классификатор мы единожды используем для поиска новых вредоносных доменов на том же графе. После чего процесс повторяется для новой трехчасовой последовательности $querylog$.

Фундаментальный принцип, заложенный в процесс обучения классификатора - это кросс-валидация. Далее, я буду понимать под классификатором двойку $(clf, param) \in (CLF, PARAM)$.

Выбирается самый «стабильный» классификатор относительно k различных произвольных разбиений обучающей выборки (напомню, что мы разбиваем на 4 части), таким образом получается самый «качественный» вариант (метрика ROC_AUC удобна для сравнения классификаторов между собой). Слово «стабильный» следует понимать так: для любого из исходных разбиений выборки, классификатор работает достаточно «качественно»

Принцип, заложенный в шаги 5 – 6 называется $GridSearch$. Обучение классификатора - это некоторая задача оптимизации. Тут же мы строим оптимизационную задачу на более высоком уровне. Мы оптимизируем по всем классификаторам, максимизируя «качество» классификации. Эта «метазадача» (формула (6.3)) позволяет выбрать наилучший классификатор из представленных.

7 Техническая реализация

В этой главе речь пойдет непосредственно о реализации алгоритмов, предложенных в данной работе и использовании их в реальной жизни. В качестве источника *querylog* будем использовать DNS сервера компании SkyDNS.

7.1 Инфраструктура сбора *querylog*

На рисунке 7.1 ниже изображен процесс сбора логов с серверов.

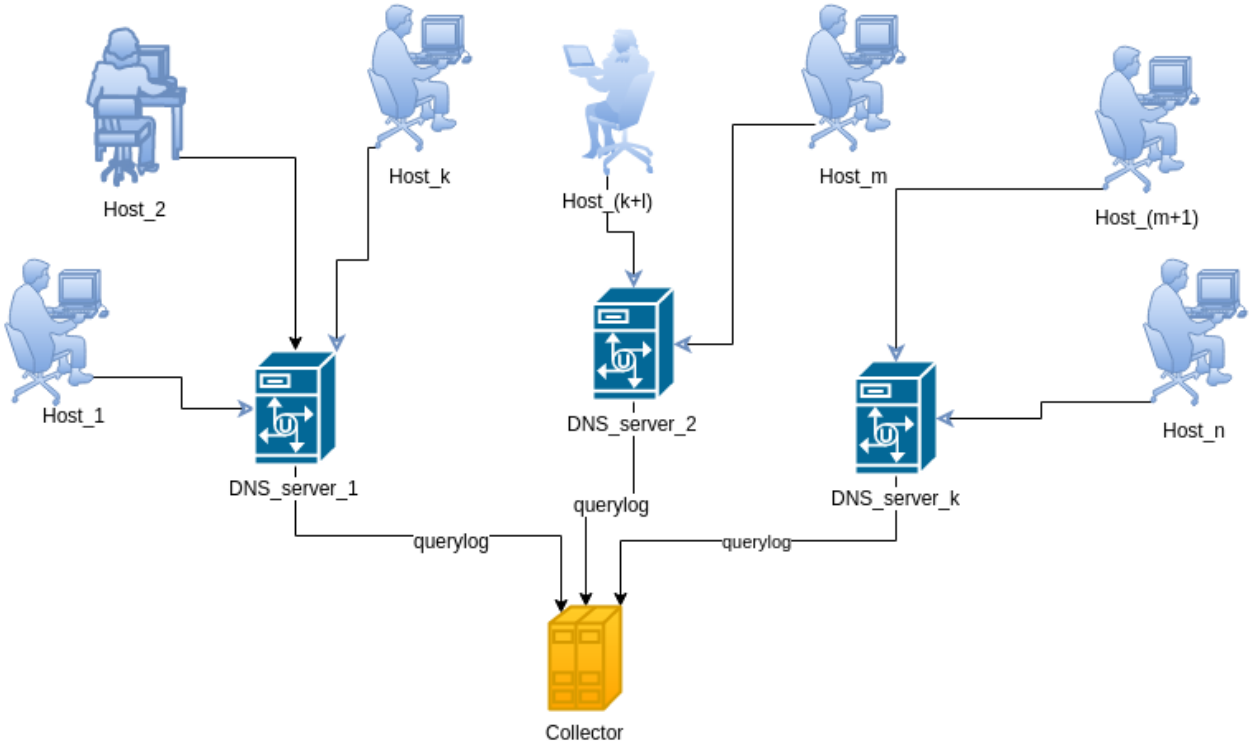


Рисунок 7.1 – Сбор логов с DNS-серверов

Пользователи DNS серверов обозначены как $host_i$, DNS сервера обозначены как DNS_server_j. Таких серверов около 10 штук. Collector - это сервер, на который собираются *querylog* и на котором запускаются алгоритмы их обработки. Этот сервер раз в сутки соединяется со всеми DNS серверами и «выкачивает» новые *querylog* (всё это делает обычный скрипт на bash с помощью rsync).

Важно отметить, что в текущей архитектуре DNS сервер состоит из 2 частей: сам DNS сервер (Bind) и прокси, который стоит перед ним и пропускает через себя все запросы пользователей. Это нужно для того, чтобы

реализовывать персональную фильтрацию для каждого из пользователей. Для этого в DNS пакет добавляется токен, которому соответствуют конкретные настройки фильтрации.

Важно, что именно прокси записывает необходимые нам *querylog* на диск. Эту задачу нельзя переложить на Bind, так как при включении логгирования запросов время его ответа увеличивается нелинейно, в зависимости от количества запросов. Следовательно, при большом количестве запросов, он существенно замедляется, что сказывается на комфорте пользователей, а это неприемлемо. Подтверждение данных слов можно найти в презентации [8].

Также, прокси записывает дополнительную информацию (например, profile_id пользователя и категории, которые присвоены запрашиваемому домену). Всю эту информацию Bind не может логгировать по техническим причинам.

7.2 Процесс обработки *querylog*

Далее начинается процесс обработки *querylog*, который подробно изображён на рисунке 7.2. Эта схема позволяет увидеть целостную картину, объединяющую подходы, описанные в данной работе.

В синей зоне изображены части *querylog*, которые Collector собрал с DNS серверов.

В красной зоне происходят общие операции, необходимые для обоих подходов. Вначале происходит группировка *querylog* на непрерывные трехчасовые блоки по каждому серверу, на каждую такую группу впоследствии будут применены все остальные операции. Далее *querylog* разбираются (из них извлекаются $client\_ip, dname$ и другие необходимые поля), после чего строятся графы, описанные в главе 5.

В желтой зоне происходит процесс обработки, описанный в разделе 5.4. В текущий момент, этот процесс не является полностью автоматизированным, так как он требует инфраструктуры для сбора дополнительной информации (признаков), которые требуются классификатору. На текущий момент, из этой зоны мы можем получить лишь плотные подграфы, которые рассматриваются вручную. Мы можем фильтровать эти подграфы по

содержанию в них вредоносных доменов, тем самым существенно уменьшив их количество. Для нас такой подграф дает информацию о новых вредоносных доменах, о пользователях, которые находятся в одном ботнете а также раскрывает часть структуры ботнета.

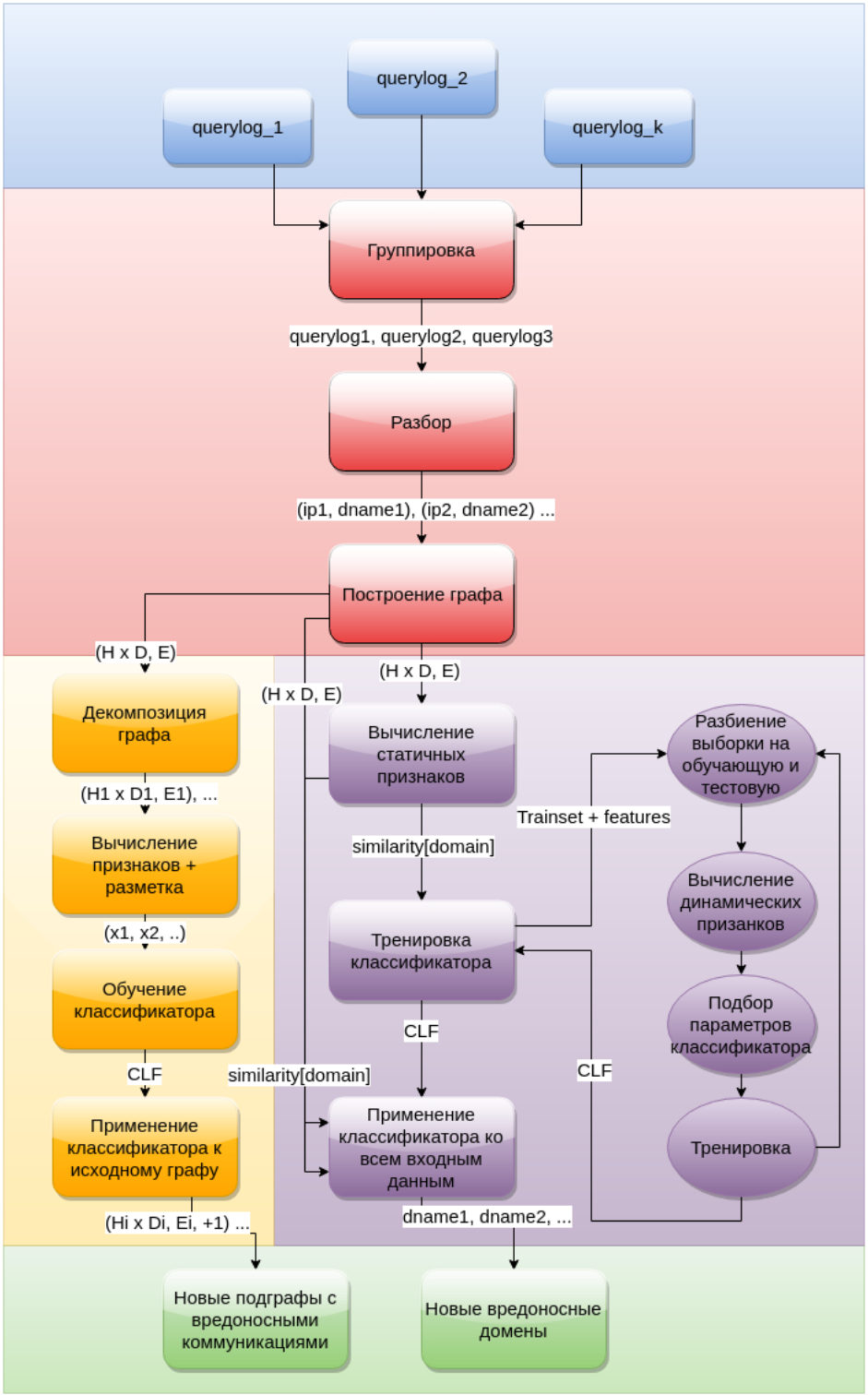


Рисунок 7.2 – Процесс обработки и анализа *querylog*

В фиолетовой зоне происходит построение классификатора для поиска вредоносных доменов. Данный процесс подробно расписан в главе 6.

В зеленой зоне содержатся результаты работы всех алгоритмов. В одном случае, это вредоносные подграфы, в другом же - доменные имена. В компании SkyDNS эта информация используется для блокировки доменных имен (на этапе разрешения адреса DNS сервером), что позволяет уберечь пользователя от невольного участия в DDoS атаке. Кроме того, данная информация позволяет уведомлять пользователей о вредоносной активности на их ПК. Кроме того, информация полезна антивирусным компаниям, так как именно они занимаются обнаружением и уничтожением ботнетов (что есть достаточно трудоемкий процесс).

7.3 Описание скрипта для поиска вредоносных доменных имен

Данный скрипт (реализующий подход, описанный в главе 6) написан на языке python версии 2.7, реализован в виде консольной утилиты, принимающей на вход параметры, описанные в таблице 7.1.

Таблица 7.1 – Ключи для запуска скрипта

Ключ	Параметр	Описание
–help		Показать справку
–files	[filename [filename ...]]	Список *querylog* файлов для обработки
–blacklist	filename	Список заведомо вредоносных доменов
–whitelist	filename	Список заведомо чистых доменов
–output	filename	Выходной файл с предсказаниями
–n_folds	number	Количество разбиений выборки при кроссвалидации
–n_iter	number	Количество итераций при просчете рангов доменов
–verbose		Отладочный вывод

В среднем, полная обработка 3 часов *querylog* с помощью этого скрипта

занимает 30 минут (варьируется в зависимости от количества запросов к DNS серверам в пределах от 10 до 40 минут). Каждый день, алгоритм находит около 5 тысяч новых вредоносных доменов. Исходный код данного скрипта содержится в приложении А.

7.4 Описание скрипта для поиска вредоносных подграфов

Данный скрипт также реализован на языке python в формате IPython Notebook. У этого формата есть ряд преимуществ, например, быстрое изменение исходного кода, работа в браузере и возможность «пересчета» отдельных блоков кода. Настоящий подход достаточно удобен, так как вредоносные подграфы используются для расследования инцидентов, связанных с подозрительной активностью групп пользователей. Так как процесс анализа подграфов, на текущий момент, не является автоматическим и требует человеческого вмешательства, поэтому, подобная реализация является уместной в рамках этой работы.

В качестве входных данных скрипт использует несколько частей *querylog*. В качестве результата его работы, пользователь получает изображения подграфов, с указанием конкретных ip адресов клиентов и доменов. Далее, пользователь использует эту информацию для информирования зараженных хостов, а также для изучения структуры ботнета.

8 Результаты работы

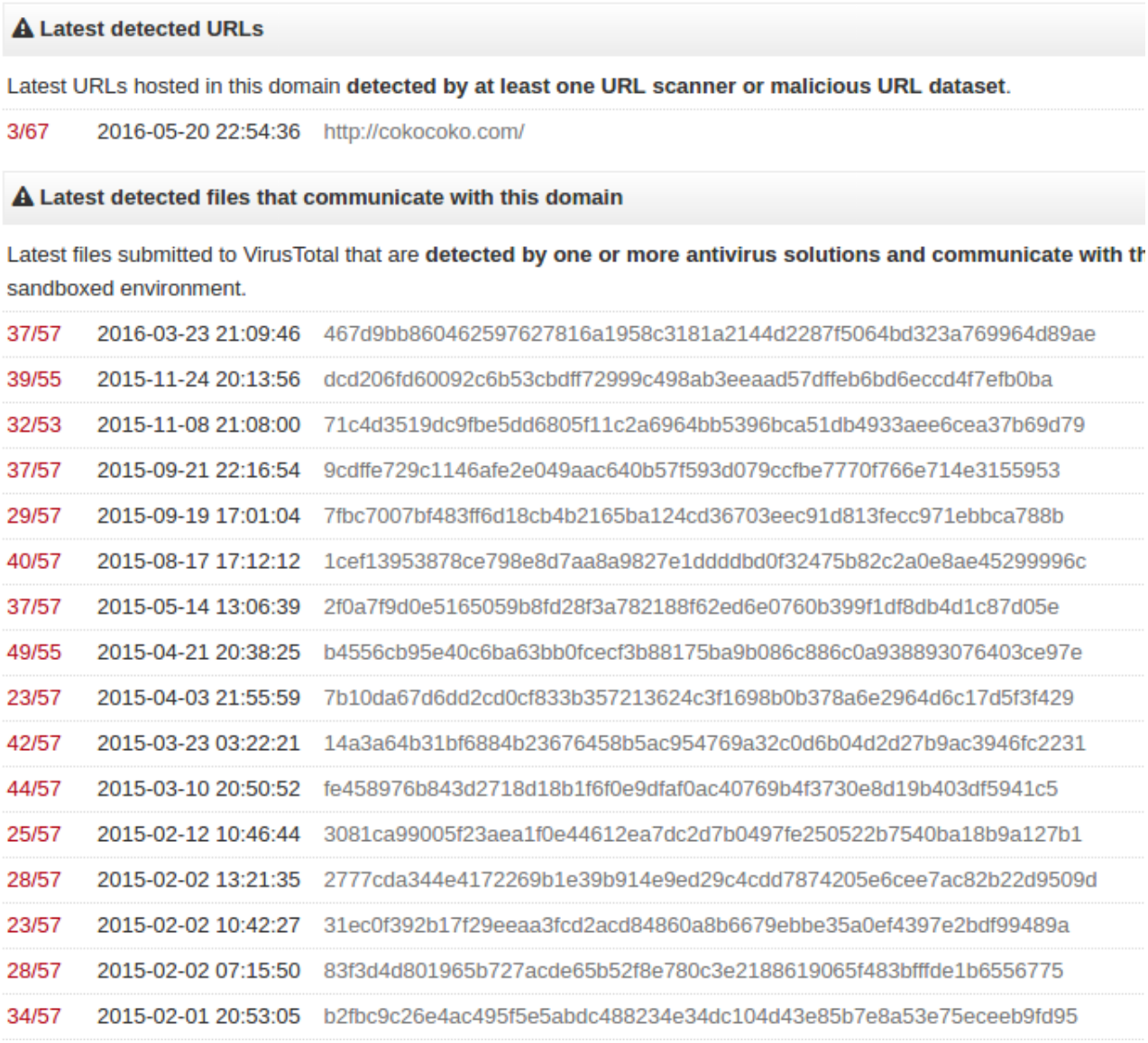
8.1 Примеры работы алгоритмов

В таблице 8.1 представлены некоторые доменные имена, которые классификатор посчитал вредоносными и «чистыми».

Таблица 8.1 – Найденные домены

Вредоносные домены	Чистые домены
gelcpzvqfrv.info	mlstat.com
ljxemtwmrmvtleqijtkppzxsdl.info	sporterr.com
cokocoko.com	garmin.com
zttwjnoftkljwchmauhytivozcqsc.com	gizmodo.com
reumatologiapalermo.it	best-torrents.net
rwlfjbbqprgtgpfecascxcpblv.org	dotcime.com
puauuanghclundcrmo.ug	btzoo.eu
wadufh.net	reference.com
ysoiigwu.com	motherlessmedia.com
fersen.it	amazon.nl
vajjxnx.org	6pm.com
nvfutv.info	diapers.com
dpjtdw.net	pofqm.xyz
kupqmwb.net	hipersushiads.com
jhypyvaqjtiohfn.com	gmtdmp.com
jhmstfhi.net	porntrex.com
okchjegutoewbuojl.ir	googledrive.com
yqodilbwxeubihahqtjdapm.org	mdotm.com
dtbkjassn.org	fuckmaturesex.com
mpreaupelxd.ru	telegraph.co.uk

Рассмотрим, например, домен cokocoko.com, проверим, что про него знает virustotal.com.



⚠ Latest detected URLs

Latest URLs hosted in this domain **detected by at least one URL scanner or malicious URL dataset**.

3/67	2016-05-20 22:54:36	http://cokocoko.com/

⚠ Latest detected files that communicate with this domain

Latest files submitted to VirusTotal that are **detected by one or more antivirus solutions and communicate with th** sandboxed environment.

37/57	2016-03-23 21:09:46	467d9bb860462597627816a1958c3181a2144d2287f5064bd323a769964d89ae
39/55	2015-11-24 20:13:56	dcd206fd60092c6b53cbdff72999c498ab3eeaad57dffeb6bd6eccd4f7efb0ba
32/53	2015-11-08 21:08:00	71c4d3519dc9fbe5dd6805f11c2a6964bb5396bca51db4933aee6cea37b69d79
37/57	2015-09-21 22:16:54	9cdffe729c1146afe2e049aac640b57f593d079ccfbe7770f766e714e3155953
29/57	2015-09-19 17:01:04	7fbc7007bf483ff6d18cb4b2165ba124cd36703eec91d813fecc971ebbca788b
40/57	2015-08-17 17:12:12	1cef13953878ce798e8d7aa8a9827e1ddddbd0f32475b82c2a0e8ae45299996c
37/57	2015-05-14 13:06:39	2f0a7f9d0e5165059b8fd28f3a782188f62ed6e0760b399f1df8db4d1c87d05e
49/55	2015-04-21 20:38:25	b4556cb95e40c6ba63bb0fcecf3b88175ba9b086c886c0a938893076403ce97e
23/57	2015-04-03 21:55:59	7b10da67d6dd2cd0cf833b357213624c3f1698b0b378a6e2964d6c17d5f3f429
42/57	2015-03-23 03:22:21	14a3a64b31bf6884b23676458b5ac954769a32c0d6b04d2d27b9ac3946fc2231
44/57	2015-03-10 20:50:52	fe458976b843d2718d18b1f6f0e9dfaf0ac40769b4f3730e8d19b403df5941c5
25/57	2015-02-12 10:46:44	3081ca99005f23aea1f0e44612ea7dc2d7b0497fe250522b7540ba18b9a127b1
28/57	2015-02-02 13:21:35	2777cda344e4172269b1e39b914e9ed29c4cdd7874205e6cee7ac82b22d9509d
23/57	2015-02-02 10:42:27	31ec0f392b17f29eeaa3fcd2acd84860a8b6679ebbe35a0ef4397e2bdf99489a
28/57	2015-02-02 07:15:50	83f3d4d801965b727acde65b52f8e780c3e2188619065f483bfffde1b6556775
34/57	2015-02-01 20:53:05	b2fbc9c26e4ac495f5e5abdc488234e34dc104d43e85b7e8a53e75eceeb9fd95

Рисунок 8.1 – Отчёт virustotal.com по домену cokocoko.com

Как видно на рисунке 8.1, этот домен действительно вредоносный. К нему обращалось множество модификаций вируса (это видно по SHA256 суммам от исполняемых файлов, которые взаимодействовали с этим доменом).

С помощью домена rwlfjbbqprgtgpfecascxcpblv.org было найдено приложение к отчету ФБР [9], связаному с наблюдением за ботнетом Zeus.

Что касается подхода с поиском сообществ, чаще всего встречаются тривиальные подграфы (две вершины и одно ребро), ощутимо реже встречаются более интересные конфигурации («звезды» разных типов, деревья, подробно это описано в разделе 5.4).

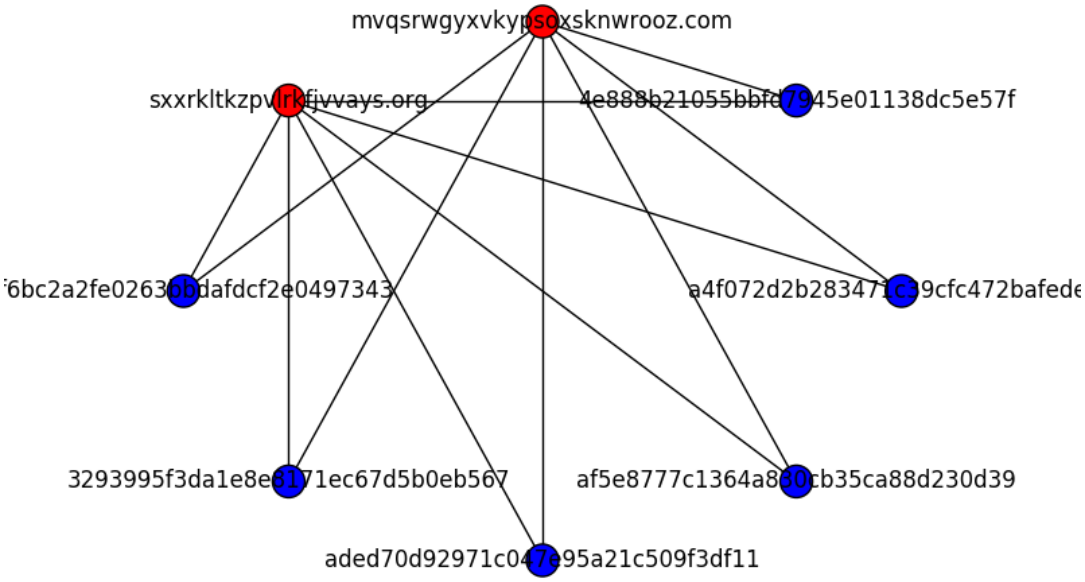


Рисунок 8.2 – Вредоносный подграф

На рисунке 8.2 изображен один из «интересных» подграфов. Красным цветом отмечены вершины, соответствующие доменным именам, синим - hash+salt от ip адресов пользователей (по причине того, что недопустимо раскрывать реальные адреса пользователей).

8.2 Оценка работы классификатора для доменных имен

Так как были использованы техники машинного обучения и построены классификаторы (построение описано в главе 6), стоит обратить внимание на формальные метрики оценки качества обученных классификаторов.

Естественно, все эти метрики вычисляются каждый раз во время процесса построения новых классификаторов и именно по ним принимается решение о выборе наилучшего экземпляра. За счет оценки по метрикам достигается автоматизация процесса обучения. Для демонстрации, я приведу небольшой пример ниже.

Возьмем случайный набор из 10 построенных классификаторов и выпишем значения основных метрик в таблицу 8.2.

Таблица 8.2 – Значение формальных метрик для набора классификаторов

TP	FP	TN	FN	precision	recall	roc-auc
874,0000	9.5000	15702.2500	38.5000	0.9893	0.9578	0.9929
798.0000	3.7500	7040.0000	27.7500	0.9953	0.9664	0.9962
849.5000	0.7500	14750.0000	29.5000	0.9991	0.9664	0.9943
887.7500	4.5000	14272.7500	26.2500	0.9950	0.9713	0.9936
861.5000	2.7500	14080.2500	43.7500	0.9968	0.9517	0.9914
847.5000	6.7500	17723.0000	41.2500	0.9921	0.9536	0.9937
544.0000	7.7500	6964.7500	41.5000	0.9860	0.9291	0.9953
949.2500	5.7500	20565.7500	45.5000	0.9940	0.9543	0.9924
797.2500	4.0000	15613.2500	34.5000	0.9950	0.9585	0.9919
968.5000	3.5000	14656.2500	30.7500	0.9964	0.9692	0.9932

Данные метрики подтверждают качество построенных классификаторов и, формально, говорят о пригодности использования описанного подхода в реальной жизни.

ЗАКЛЮЧЕНИЕ

В результате данной работы были изучены многочисленные статьи, произведен анализ их содержимого на предмет практической применимости. В сравнении с другими предложениями, текущее решение обладает важными качествами:

1. Не требует труднодоступной информации (например данных pDNS и WHOIS).

2. Не требует реверс-инжиниринга кода или запуска семплов вирусов в песочнице для анализа. Этот процесс всегда достаточно трудоемок и требует непосредственного участия высококвалифицированного специалиста.

3. Процесс полностью автоматизирован и не требует вмешательства человека.

«В бою», эта техника показала себя хорошо, но в дальнейшем требуется больше внимания уделить «охвату» классификатора (минимизации ошибок 2 рода), чтобы находить больше вредоносных доменов а также автоматическому выбору порога по ROC кривой.

В данный момент, результат работы внедрен в компании SkyDNS и помогает предотвращать вредоносные коммуникации пользователей а также предупреждать их об опасности.

Задачи, поставленные во введении, достигнуты полностью. В результате анализа и работы был получен готовый программный продукт, который удовлетворяет требованиям бизнеса.

В дальнейшем, планируется развивать направление, связанное с анализом структуры графа запросов:

1. Создание инструмента, позволяющего в интерактивном режиме исследовать данный граф и искать в нем скрытые связи.

2. Реализация автоматической классификация подграфов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Gunter Ollmann, “Botnet Communication Topologies. Understanding the intricacies of botnet command-and-control”, 2009.

2. Hyunsang Choi, Hanwoo Lee, Heejo Lee, Hyogon Kim, “Botnet Detection by Monitoring Group Activities in DNS Traffic”.

3. Keisuke Ishibashi, Tsuyoshi Toyono, Makoto Iwamura, “Botnet Detection Combining DNS and Honeypot Data”.

4. Frank Denis, “Malware vs Big Data”, pp. 30 – 31.

5. Y. Jin, E. Sharafuddin, and Z. L. Zhang, “Unveiling core network-wide communication patterns through application traffic activity graph decomposition,” in Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems. ACM, 2009, pp. 49 – 60.

6. P. Mockapetris, “DOMAIN NAMES - IMPLEMENTATION AND SPECIFICATION, RFC1035”.

7. C. Ding, T. Li, W. Peng, H. Park, “Orthogonal nonnegative matrix t-factorizations for clustering”, in Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining, ser. KDD ’06. New York, NY, USA: ACM, 2006, pp. 126–135.

8. Jeroen Massar, “DNSTAP: high speed DNS logging without packet capture”, pp. 29 – 35.

9. “Appendix for FBI report about Zeus domains”, pp. 73, `https://www.justice.gov/opa/file/798036/download`.

A Скрипт для поиска вредоносных доменов

```
import argparse
import logging
import itertools
import numpy as np
from joblib import Parallel, delayed
from sklearn.cross_validation import StratifiedKFold
from sklearn.ensemble import AdaBoostClassifier, RandomForestClassifier,
    GradientBoostingClassifier
from sklearn.grid_search import ParameterGrid
from sklearn.metrics import roc_auc_score, precision_score, recall_score
from tldextract import TLDExtract

from binlog_reader import binlog_reader

logger = logging.getLogger(__name__)


class DomainFilter(object):
    def __init__(self):
        self._extract = TLDExtract(include_psl_private_domains=True)

    def transform(self, domain):
        return self._extract(domain).registered_domain


def ga_similarities(ipl_1, ipl_2):
    a, b = float(len(ipl_1)), float(len(ipl_2))
    c = float(len(ipl_1 & ipl_2))
    return 0.5 * (c / a + c / b), c / len(ipl_1 | ipl_2), int(c)


def group_activities(all_domains, domain2ip_by_hour):
    logger.debug("Calculating group activity")
    result = {domain: dict() for domain in all_domains}
    pairs = list(filter(lambda (x, y): x < y, itertools.permutations(range(len
        (domain2ip_by_hour)), 2)))
    feature_name_pattern = "({}, {})_{}"

    for idx, domain in enumerate(all_domains):
        for curr, other in pairs:
            fst_hour, snd_hour = domain2ip_by_hour[curr].get(domain),
                domain2ip_by_hour[other].get(domain)
            sim, jcd, ln = 0., 0., 0.
            if fst_hour and snd_hour:
```

```
                sim, jcd, ln = ga_similarities(fst_hour, snd_hour)

            result[domain].update({feature_name_pattern.format(curr, other, "
                sim"): sim,
                                   feature_name_pattern.format(curr, other, "
                                       jcd"): jcd,
                                   feature_name_pattern.format(curr, other, "
                                       length"): ln})
        if (idx + 1) % 100000 == 0:
            logger.debug("Processed %d domains", idx + 1)

    return result


def ranking(ip2d, d2ip, X, y, init_abs_score=10, n_iter=20):
    logger.debug("Calculating rank scores")
    rank_ip = {ip: {'sc_score': 0., 'black_score': 0., 'white_score': 0.} for
        ip in ip2d}
    rank_d = {d: {'sc_score': 0., 'black_score': 0., 'white_score': 0.} for d
        in d2ip}

    for dom, cls in zip(X, y):
        if cls == 1:
            rank_d[dom]['sc_score'] = -float(init_abs_score)
            rank_d[dom]['black_score'] = -float(init_abs_score)

        elif cls == -1:
            rank_d[dom]['sc_score'] = float(init_abs_score)
            rank_d[dom]['white_score'] = float(init_abs_score)

    for it in range(n_iter):
        logger.debug("Iteration %d", it + 1)

        for ip in rank_ip:
            rank_ip[ip]['sc_score'] = sum(rank_d[d]['sc_score'] / len(d2ip[d])
                 for d in ip2d[ip])
            rank_ip[ip]['black_score'] = sum(rank_d[d]['black_score'] / len(
                d2ip[d]) for d in ip2d[ip])
            rank_ip[ip]['white_score'] = sum(rank_d[d]['white_score'] / len(
                d2ip[d]) for d in ip2d[ip])

        for domain in rank_d:
            rank_d[domain]['sc_score'] = sum(rank_ip[ip]['sc_score'] / len(
                ip2d[ip]) for ip in d2ip[domain])
            rank_d[domain]['black_score'] = sum(rank_ip[ip]['black_score'] /
                len(ip2d[ip]) for ip in d2ip[domain])
            rank_d[domain]['white_score'] = sum(rank_ip[ip]['white_score'] /
                len(ip2d[ip]) for ip in d2ip[domain])
```

```
    return rank_d


def read_logfile(fname, fields=("client_ip", "dname")):
    with open(fname, 'rb') as infile:
        logger.debug("Open file %s", fname)
        reader = binlog_reader(infile, fields)
        return set([tuple([query[fld] for fld in fields]) for query in reader
            ])


def create_indexes(pairs):
    logger.debug("Creating indexes domain2ip & ip2domain")
    domain2ip, ip2domain = dict(), dict()
    for ip, domain in pairs:
        domain2ip.setdefault(domain, set())
        domain2ip[domain].add(ip)
        ip2domain.setdefault(ip, set())
        ip2domain[ip].add(domain)

    return domain2ip, ip2domain


def merge_indexes(indexes):
    logger.debug("Merge %d indexes to one", len(indexes))
    ip2domain_full, domain2ip_full = dict(), dict()
    for (d2ip, ip2d) in indexes:
        for domain in d2ip:
            domain2ip_full.setdefault(domain, set())
            domain2ip_full[domain] |= d2ip[domain]

        for ip in ip2d:
            ip2domain_full.setdefault(ip, set())
            ip2domain_full[ip] |= ip2d[ip]
    return domain2ip_full, ip2domain_full


def create_domain_indexes(hosts):
    logger.debug("Creating indexes host2domain & domain2host")
    df = DomainFilter()
    domain2host, host2domain = dict(), dict()
    for host in hosts:
        domain = df.transform(host)
        domain2host.setdefault(domain, set())
        domain2host[domain].add(host)
        host2domain[host] = domain
```

```
    return domain2host, host2domain


def prepare_trainset(blacklist, whitelist, all_domains):
    df = DomainFilter()
    with open(blacklist, 'r') as infile:
        blacklist_domains = [df.transform(line.strip()) for line in infile]
    with open(whitelist, 'r') as infile:
        whitelist_domains = [df.transform(line.strip()) for line in infile]

    pos, neg = set(filter(lambda d: d, blacklist_domains)), set(filter(lambda
        d: d, whitelist_domains))

    inter = pos & neg
    logger.debug("Positive size %d, Negative size %d, Intersection %d, All
        domains %d",
                 len(pos), len(neg), len(inter), len(all_domains))
    pos, neg = (pos - inter) & all_domains, (neg - inter) & all_domains
    logger.debug("Positive after %d, Negative after %d", len(pos), len(neg))

    X = list(pos) + list(neg)
    y = [1 for _ in range(len(pos))] + [-1 for _ in range(len(neg))]
    return np.array(X), np.array(y)


def join_features_by_keys(keys, features_list):
    logger.debug("Joining features")
    tmp_full = {key: dict() for key in keys}
    for domain in tmp_full:
        for features in features_list:
            tmp_full[domain].update(features[domain])

    all_features = sorted(tmp_full[keys[0]].keys())
    logger.debug("Features - %s", ','.join(str(x) for x in all_features))
    return np.array([[tmp_full[domain][f] for f in all_features] for domain in
        keys])


def cacl_stat(y_test, out_lab):
    tp, tn, fp, fn = 0, 0, 0, 0
    for true, pred in zip(y_test, out_lab):
        if true == pred:
            if pred == 1:
                tp += 1
            else:
                tn += 1
        elif true == 1:
            fn += 1
```

```
        else:
            fp += 1

    return tp, tn, fp, fn


def learn_clf(classifier, param, X_train, y_train, X_test, y_test):
    clf = classifier(**param)
    clf.fit(X_train, y_train)
    out = clf.predict_proba(X_test)
    out_lab = clf.predict(X_test)

    if len(out.shape) == 2:
        out = out[:, np.where(clf.classes_ == 1)[0][0]]

    return classifier, param, {"roc-auc": roc_auc_score(y_test, out),
                               "tp-tn-fp-fn": cacl_stat(y_test, out_lab),
                               "precision": precision_score(y_test, out_lab),
                               "recall": recall_score(y_test, out_lab)}


def grid_search(grid, X_train, y_train, X_test, y_test):
    logger.debug("Start grid search")
    with Parallel(n_jobs=-1, backend="multiprocessing") as parallel:
        result = parallel(delayed(learn_clf)(clf, param, X_train, y_train,
            X_test, y_test)
                          for (clf, param) in grid)
    logger.debug("End grid search")
    return result


def make_predictor(X, y, ip2domain_full, domain2ip_full, const_features,
    n_folds, n_iter):
    logger.debug("Creating classifier, n_folds = %d", n_folds)
    skf = StratifiedKFold(y, n_folds=n_folds)

    clfs = [
        (AdaBoostClassifier, {"n_estimators": [30, 50, 100, 150, 200, 250,
            300],
                              "learning_rate": [1., 0.8, 0.5, 0.1, 0.05]}),
        (RandomForestClassifier, {"n_estimators": range(10, 150, 10),
                                  "criterion": ["gini", "entropy"],
                                  "max_features": ["sqrt", "log2", None]}),
        # (GradientBoostingClassifier, {"learning_rate": [0.07, 0.1, 0.3],
        #                               "n_estimators": [50, 100, 200],
        #                               "max_depth": range(2, 5)})
    ]
```

```
grid = [(clf, param) for clf, parameters in clfs for param in
    ParameterGrid(parameters)]
full_scores = {(clf, frozenset(param.items())): {"roc-auc": [],
                                                 "tp-tn-fp-fn": [],
                                                 "precision": [],
                                                 "recall": []} for (clf,
                                                     param) in grid}

for idx, (train_index, test_index) in enumerate(skf):
    logger.debug("Folding iteration #%d", idx + 1)
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]
    rank_features = ranking(ip2domain_full, domain2ip_full, X_train,
        y_train,
                            init_abs_score=10, n_iter=n_iter)

    X_features_train = join_features_by_keys(X_train, [const_features,
        rank_features])
    X_features_test = join_features_by_keys(X_test, [const_features,
        rank_features])

    res = grid_search(grid, X_features_train, y_train, X_features_test,
        y_test)

    for clf, param, score in res:
        for metric in score:
            full_scores[(clf, frozenset(param.items()))][metric].append(
                score[metric])

final_score = [(clf_class, params, {"roc-auc": np.mean(full_scores[(
    clf_class, params)]["roc-auc"]),
                                    "precision": np.mean(full_scores[(
                                        clf_class, params)]["precision"]),
                                    "recall": np.mean(full_scores[(
                                        clf_class, params)]["recall"]),
                                    "TP": np.mean([x[0] for x in
                                        full_scores[(clf_class, params)]["
                                        tp-tn-fp-fn"]]),
                                    "TN": np.mean([x[1] for x in
                                        full_scores[(clf_class, params)]["
                                        tp-tn-fp-fn"]]),
                                    "FP": np.mean([x[2] for x in
                                        full_scores[(clf_class, params)]["
                                        tp-tn-fp-fn"]]),
                                    "FN": np.mean([x[3] for x in
                                        full_scores[(clf_class, params)]["
                                        tp-tn-fp-fn"]])})
               for (clf_class, params) in full_scores]
```

```
    final_score.sort(key=lambda _: _[2]["roc-auc"], reverse=True)

    logger.info("Top 50 params")
    for idx, (c, p, s) in enumerate(final_score[:50]):
        logger.debug("# %d | Clf: %s, params: %s, scores: %s", idx + 1, c.
            __name__, str(p), str(s))

    final_clf = final_score[0][0](**dict(final_score[0][1]))
    return final_clf


def processing(logfiles, blacklist, whitelist, output_file, n_folds, n_iter):
    queries = [read_logfile(fn, ("client_ip", "dname")) for fn in sorted(
        logfiles)]
    hosts = set([domain for (ip, domain) in itertools.chain.from_iterable(
        queries)])
    domain2host, host2domain = create_domain_indexes(hosts)

    queries = [[(ip, host2domain[domain]) for (ip, domain) in hour] for hour
        in queries]
    queries = [set(filter(lambda (_, dom): dom, hour)) for hour in queries]

    small_indexes = [create_indexes(hour) for hour in queries]
    domain2ip_full, ip2domain_full = merge_indexes(small_indexes)
    all_domains = set(domain2ip_full.keys())
    domain2ip_by_hour = [d2ip for (d2ip, ip2d) in small_indexes]

    X, y = prepare_trainset(blacklist, whitelist, all_domains)
    ga_features = group_activities(all_domains, domain2ip_by_hour)
    clf = make_predictor(X, y, ip2domain_full, domain2ip_full, ga_features,
        n_folds, n_iter)

    rank_final_features = ranking(ip2domain_full, domain2ip_full, X, y,
        init_abs_score=10, n_iter=n_iter)
    X_final = join_features_by_keys(X, [ga_features, rank_final_features])
    clf.fit(X_final, y)

    all_domains = list(all_domains)
    X_full = join_features_by_keys(all_domains, [ga_features,
        rank_final_features])
    result_prob, result_bin = clf.predict_proba(X_full)[:, np.where(clf.
        classes_ == 1)[0][0]], clf.predict(X_full)

    labeled_domains = {x: lab for x, lab in zip(X, y)}

    to_file = sorted(zip(all_domains, result_prob, result_bin), key=lambda x:
        x[1], reverse=True)
```

```
    logger.debug("Save result to %s", output_file)
    with open(output_file, 'w') as outfile:
        for d, prob, lab in to_file:
            in_train = labeled_domains.get(d, 0)
            outfile.write("{}\t{}\t{}\t{}\t{}\n".format(d, prob, lab, ",".join
                (domain2host[d]), in_train))


def main():
    parser = argparse.ArgumentParser(description="Suspicious domain detector (
        used querylog)")

    parser.add_argument('-f', '--files', help='Files with logs', required=True
        , nargs='*')
    parser.add_argument('-b', '--blacklist', help='Path to blacklist',
        required=True, type=str)
    parser.add_argument('-w', '--whitelist', help='Path to whitelist',
        required=True, type=str)
    parser.add_argument('-o', '--output', help='Path to output prediction',
        required=True, type=str)
    parser.add_argument('-v', '--verbose', help='Verbose flag', action='
        store_const', dest="loglevel",
                        const=logging.DEBUG, default=logging.WARNING)
    parser.add_argument('--n_folds', help='Number of folds in cv stage',
        default=4, type=int)
    parser.add_argument('--n_iter', help='Number of iteration for rank calc',
        default=20, type=int)
    args = parser.parse_args()
    logging.basicConfig(format='%(asctime)s - %(levelname)s - %(message)s',
        level=args.loglevel)
    return processing(args.files, args.blacklist, args.whitelist, args.output,
                      args.n_folds, args.n_iter)


if __name__ == '__main__':
    exit(main())
```