Московский государственный университет
имени М. В. Ломоносова

факультет вычислительной математики и кибернетики
Лаборатория интеллектуальных систем кибербезопасности

Курсовая работа

Биометрический фингерпринтинг:
идентификация пользователей по особенностям использования внешних устройств ввода

Важдаев А. С.
321 группа
Научный руководитель к.ф.-м.н., с.н.с., Гамаюнов Д. Ю.

Аннотация

В работе выдвигается гипотеза о возможности идентификации клиента по особенностям его взаимодействия с внешними устройствами ввода и строится модельная система для проверки данной гипотезы.

Содержание

1 Введение

В современном Интернете широко распространено такое явление, как фингерпринтинг (от англ. fingerprint — отпечаток пальца). В общих чертах, это набор определённых техник, позволяющих идентифицировать пользовательское устройство без использования каких-либо специализированных хранилищ данных (Cookies, LSO, кэш браузера и т. п.). Эти техники основаны на получении (как правило, через браузерный JavaScript) значений различных параметров машины, таких как версия браузера, набор установленных шрифтов, разрешение экрана и т. д. Каждый из этих параметров в отдельности малозначителен, но их совокупность позволяет идентифицировать пользователей с вероятностью, близкой к 100%, а способы фингерпринтинга постоянно совершенствуются.

Как правило, техники фингерпринтинга используются совместно с традиционными методами идентификации (например, через файлы Cookie), и вместе они позволяют гарантированно определять клиентское устройство.

2 Причины и постановка задачи

2.1 Проблемы «железного» фингерпринтинга

Фингерпринтинг различного характера повсеместно используется для идентификации клиентских устройств в самых различных целях — от защиты клиента от несанкционированного проникновения до тотального шпионажа за поведением пользователя в Сети. Но, какими бы ни были эти методы, у них есть один большой недостаток — они всецело ориентированы на определение именно физического устройства клиента.

Проблемы у такого рода фингерпринтинга возникают в двух противоположных ситуациях. Первая — когда один человек меняет свои отпечатки. Это может происходить в следующих случаях:

- Клиент использует несколько разных устройств (ноутбук, рабочий ПК, телефон, планшет и т. д.);
- Клиент использует различное ПО на одном устройстве (разные браузеры);
- Клиент модифицирует текущее ПО (новые версии браузера, установка и обновление расширений).

Такие ситуации во многих случаях «собьют с толку» систему фингерпринтинга. Другая ситуация ровно обратная — когда одним и тем же устройством, часто даже одним и тем же приложением (общий учебный компьютер, семейный ПК и т. д.), пользуются разные люди. Возникающие проблемы тут очевидны: если фингерпринтинг используется, например, для показа таргетированной рекламы,

то такая система не сможет правильно сориентироваться: скажем, если компьютер совместно используют родитель и ребёнок, то ребёнку может быть выдан нежелательный контент «только для взрослых».

В связи с этим возникает вопрос — можно ли научить систему фингерпринтинга различать не «железные» устройства и конкретные экземпляры и версии ПО, а самих людей? И если да, то как?

Для ответа на этот вопрос нужно подумать, как же вообще взаимодействует человек с машиной и её программным обеспечением. Ответ — с помощью внешних устройств. Отсюда вывод: если такую систему и можно построить, то она должна собирать данные с помощью этих самых устройств.

Существует идея, что можно идентифицировать пользователя по отдельным особенностям его работы с устройствами ввода (такими как клавиатура или мышь); среди таких особенностей могут быть, например, долгота нажатия клавиш на клавиатуре, скорость и «размах» движений мыши и многие другие, на первый взгляд, ничем не примечательные вещи. Всё это в некотором смысле напоминает почерковедение — криминалистическую науку, занимающуюся идентификацией почерка человека по определённому набору характерных черт и параметров.

2.2 Цель работы

Целью данной работы является проверка гипотезы о вышеописанной системе. Сформулируем эту гипотезу более формально:

«Возможно построить систему идентификации пользователей на основе данных с внешних устройств, которая будет с приемлемой эффективностью решать следующие задачи:

- Различение отдельных клиентов, пользующихся одним устройством;
- Определение различных устройств, принадлежащих одному клиенту».

Гипотезу будем проверять экспериментальным образом — построим тестовую модель такой системы, смоделируем естественные процессы взаимодействия клиентов с устройствами и проанализируем полученные данные.

3 Построение модели

Для начала, следует более точно сформулировать, какую систему мы собираемся создать. Итак, результатом данной работы должен стать программный продукт, умеющий в том числе:

- Собирать определённую информацию с внешних устройств клиента;
- Сохранять и систематизировать эту информацию;
- Анализировать имеющиеся данные и на их основе с достаточной точностью идентифицировать пользователей, например, определённого веб-сайта.

3.1 Принцип действия системы

Как и большинство методик фингерпринтинга, наша система будет использовать возможности JavaScript, а, точнее, *события*, порождаемые браузером. Среди них, в частности, есть события, связанные с внешними устройствами ввода (например, движение мыши или нажатие клавиши).

Наша система будет обрабатывать такие события, сохранять, систематизировать и анализировать, на выходе получая определённый уникальный идентификатор пользователя (или множество таких идентификаторов, если возможных вариантов несколько).

3.2 Общая схема реализации

Модельная версия нашей системы будет основана на клиент-серверной модели. Основной объём работы будет возложен на сервер — клиентская часть программы будет лишь отслеживать и отправлять на сервер поступающие события. Такая модель была выбрана в целях защиты от подделки отпечатка (подробно этот момент разобран в п. 3.6), и для упрощения самой модели.

Клиентская часть будет реализована в виде небольшого скрипта, подключающего на нужные события специальную функцию-обработчик. Эта функция будет сериализовывать данные о событии и отправлять их на сервер. Отправлять события на сервер можно, например, обычным AJAX-запросом.

Серверная часть нашей системы состоит из двух частей. Первая — функциональное ядро, которое будет, собственно, выполнять всю работу, а вторая — небольшой API, который получает запросы на сервер и отправляет их ядру, после чего выдаёт ответ ядра на запрос.

Серверная часть модельной системы реализована на языке PHP ввиду его повсеместной поддержки и простоты установки.

3.3 Клиентская часть

Итак, мы описали, каким образом будут передаваться и обрабатываться параметры; опишем теперь и сами параметры.

Как было упомянуто, методика идентификации основывается на *событиях*, порождаемых при взаимодействии пользователя с внешними устройствами. Что же это за устройства и как с ними работать?

Какие вообще есть внешние устройства, с которыми так или иначе может взаимодействовать пользователь? Перечислим некоторые из них:

- Устройства прямого ввода:
  - Клавиатура;
  - Мышь (трекпад, трекбол и т. п.);
  - Сенсорный экран;
- Медиа-устройства:
  - Микрофон;
  - Веб-камера;
- Датчики внешней среды:
  - Акселерометр, гироскоп;
  - Дактилоскоп, барометр, компас, навигация, радио и т. д.

Из всех этих устройств мы можем работать с устройствами ввода, медиа-устройствами и с некоторыми датчиками (акселерометр, навигация).

Какие из них можно использовать для идентификации?

Самый простой способ — это, конечно, просто включить веб-камеру и сфотографировать пользователя. Однако, *фингерпринтинг* в большинстве случаев должен работать **незаметно** для пользователя, а работа с камерой требует обязательного подтверждения со стороны пользователя; то же самое касается микрофона и местоположения. Акселерометр можно использовать без разрешения, но он, во-первых, доступен только на мобильных устройствах, а, во-вторых, может сказать хоть что-то о человеке только в очень узком круге ситуаций. Хотя этот параметр всё же можно использовать, ввиду высокой технической сложности реализации, мы пока не будем этого делать.

Итак, остались устройства ввода — клавиатура, мышь и сенсорный экран. Работать с ними довольно просто — каждое нажатие клавиши, экрана или перемещение мыши порождает событие, содержащее все необходимые параметры.

3.3.1 События мыши и клавиатуры

JavaScript позволяет нам максимально точно считывать работу пользователя с клавиатурой и мышью. Разберём этот момент подробнее.

Для начала, клавиатура. JavaScript может реагировать на три типа возникающих событий: `onkeydown`, `onkeyup` и `onkeypress`. Первое порождается в момент нажатия клавиши, соответственно, второе — при её отпускании. Третий тип событий возникает также либо при нажатии клавиши, либо при её отпускании, но для разных клавиш этот момент может отличаться, а на некоторые клавиши такое событие вообще не порождается; поэтому, нас этот тип событий не интересует и мы будем работать только с `onkeydown` и `onkeyup`. В таком событии, помимо его типа (нажата клавиша или отпущена), также указывается *код клавиши* — целое число, определяющее конкретную клавишу на любой клавиатуре.

С мышью связано гораздо больше различных событий, но из них нас будут интересовать лишь четыре — `onmousemove` и `onscroll`, а также `onmousedown` и `onmouseup`. Первое возникает, как видно из названия, при движении мыши; в этом событии нам понадобятся *координаты курсора* — они содержатся в глобальных переменных JavaScript `clientX` и `clientY`. Второе же порождается при прокрутке страницы — здесь нас интересует *сдвиг* — число, показывающее, сколько пикселей страницы сейчас прокручено; оно находится в глобальной переменной `pageYOffset`. Последние два события отвечают за, соответственно, нажатие и отпускание кнопок мыши. В этих событиях имеется параметр `which`, который содержит код нажатой кнопки, но мы пока не будем обрабатывать этот параметр.

3.3.2 Хранение и передача параметров

Итак, все вышеуказанные данные будут считываться фунцкией-обработчиком и отправляться на сервер. В зависимости от реализации, можно отправлять либо каждое событие в отдельности (что повышает точность, но увеличивает нагрузку на сервер и сетевой канал), либо накапливать в отдельном массиве и периодически его отправлять; для тестовой модели используется первый вариант, поскольку нам нужно иметь возможность наблюдать результаты даже одного события, но впоследствии изменить этот параметр не представляет трудности.

Но как серверу работать с этими данными? Если нам понадобится, например, вычислить среднюю скорость набора текста, то сервер должен знать время нажатия каждой клавиши. Время поступления запроса на сервер на эту роль категорически не подходит, поскольку запросы поступают на сервер не мгновенно, иногда в другом порядке, а в отдельных случаях могут вообще быть потеряны.

Для решения этой проблемы в каждое отправляемое событие будем встраивать два служебных поля — `timestamp` и `id`. Первое, как видно из названия, будет содержать *время момента возникновения* события (а не время отправки или получения). Оно представляется в UNIX-формате времени (от 1 января 1970 года) и указано с точностью до миллисекунды; таким образом время можно получить

встроенной функцией `Date.now()`.

Поле `id` используется для сохранения порядка и отслеживания потери пакетов. Оно содержит порядковый номер отправляемого события **данного типа** (т. е., для нажатий клавиш, движений мыши и других событий будет несколько независимых счётчиков). В случае потери пакетов сервер сможет попытаться восстановить усреднённые данные, например, простой линейной интерполяцией.

Обобщая вышесказанное, приведём пример функции-обработчика клиентского скрипта на примере нажатий клавиш (рис. 1).

```
function keyboardEventHandler(e){
    data = {
        'type': 'keyboard',
        'id': keyboard_id++,
        'action': (e.type == 'keydown' ? 'down' : 'up'),
        'code': e.keyCode,
        'timestamp': Date.now()
    };
    sendRequest(data); // Функция отправки запроса на сервер
}
```

Рис. 1: Функция-обработчик нажатий клавиш

3.4 Серверная часть

Итак, данные о событиях сформированы и отправлены клиентом. Разберём, что будет сервер делать дальше с этими данными.

Как было описано выше, серверная часть состоит из двух модулей — API для отправки запросов, и собственно функциональной части.

3.4.1 API обработчика событий

API модельной системы реализован на основе обычных HTTP-запросов. Информация передётся через POST-заголовки запроса; в ответ сервер выдаёт результаты своей работы:

- Если удалось однозначно идентифицировать пользователя, возвращается его ID — уникальный номер записи в базе;
- Если с приблизительно равной вероятностью оказались возможными несколько вариантов, выдаются все варианты или сообщение о необходимости сбора большего объёма данных;

- Если точно установлено, что соответствующий пользователь в базе отсутствует, то он создаётся и возвращается ID вновь созданной записи;
- В тестовой модели также возвращается *отладочная информация*: она представляет собой массив записей, состоящих из двух полей: `user` — ID пользователя, и `p` — степень соответствия этого пользователя полученным данным (число типа `float` от 0 до 1, где 1 соответствует точному совпадению).

3.4.2 Общее устройство ядра

Ядром мы будем называть модуль системы, ответственный за хранение и обработку данных. Оно также состоит из нескольких частей:

- Регистр текущего состояния — массив-буфер, в котором хранятся все события, относящиеся к данному клиенту;
- База данных (MySQL) — таблица, содержащая отпечатки всех пользователей, по которым осуществляется поиск;
- Вектор предположений — массив уровней соответствия пользователей вида `{"user": 1337, "p": 0.41452281488}` (см. выше);
- Функция анализа — вычисляет определённые параметры на основе событий, хранящихся в регистре (именно эти параметры впоследствии будут сохранены в базе данных) и формирует из них *массив параметров*;
- Функция измерения — по двум массивам параметров вычисляет «разницу» между ними (о способах её вычисления см. в п. 3.4.6);
- Функция построения вектора — сравнивает текущий массив параметров с указанными в базе и строит *вектор предположений*;
- Функция подтверждения — в случае однозначной идентификации пользователя производит самообучение модели и подстройку данных в базе; в случае отсутствия клиента в базе создаёт новую запись.

Далее мы подробено разберём каждый из этих пунктов. В целом же, диаграмма взаимодействия всех частей нашей системы представлена на рис. 2.

3.4.3 Парсинг событий и регистр состояния

Как мы уже говорили, информация о событии приходит на сервер в виде POST-запроса с соответствующими заголовками. В PHP все заголовки запроса

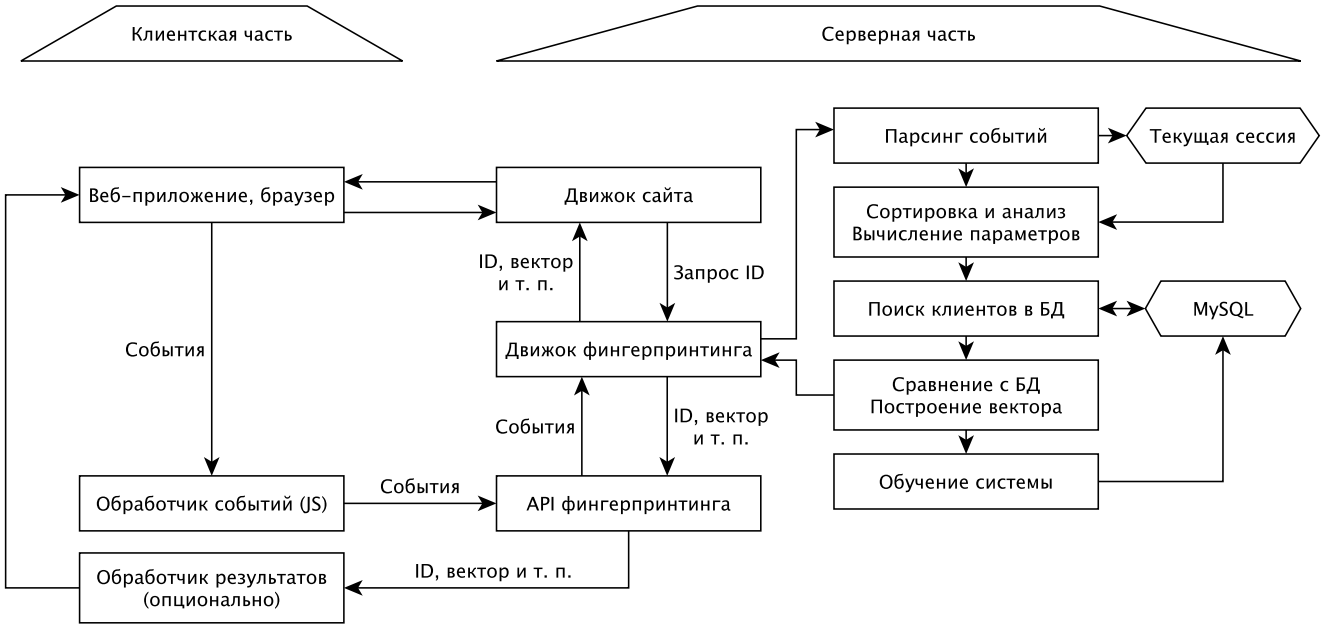

Рис. 2: Диаграмма взаимодействия модели

доступны напрямую через глобальный массив `$_POST`, поэтому в регистр состояния можно просто скопировать нужные поля этого массива. Сам регистр представляет собой массив записей, каждая из которых соответствует одному событию; порядок событий не определён, события всех типов располагаются подряд; правильный порядок обработки определяет функция анализа, регистр же просто хранит поступающие данные.

Пример фрагмента регистра состояния можно увидеть на рис. 3.

Сохранение данных между запросами. Как известно, время жизни PHP-скрипта ограничено обработкой одного запроса. Это значит, что, после того, как сервер принял очередное событие и выдал какой-либо ответ, все его переменные, функции, объекты перестают существовать. Это значит, что мы не можем просто хранить нужные данные как переменные. Для решения этой проблемы в PHP были придуманы *сессии*. Они работают на основе файлов Cookie, и представляют собой особым образом организованное файловое хранилище, куда можно записать любые переменные для какого-либо пользователя, после чего их считать. В нашей системе сессии используются для сохранения состояния объекта между запросами с одной и той же страницы; использование Cookie-файлов здесь не противоречит идее фингерпринтинга, поскольку они не используются для идентификации пользователя вообще, а лишь в рамках работы одной страницы; при желании, однако, и такой механизм можно заменить на любой, более удобный читателю.

3.4.4 Массив параметров и база отпечатков

Итоговый отпечаток, построенный на основе регистра событий, строится в виде *массива параметров*. Каждый параметр отражает те или иные характеристики

```
[
    {
        "type": "keyboard",
        "action": "down",
        "code": 8,
        "id": 13,
        "timestamp": 1554703666013
    },
    {
        "type": "mouse",
        "x": 256,
        "y": 333
        "id": 74,
        "timestamp": 1554703666545
    }
]
```


Рис. 3: Пример регистра состояния (в формате JSON)

поведения пользователя (например, скорость печати на клавиатуре) и хранится в виде числа — целого или с плавающей точкой.

Для хранения в базе данных такой массив приводится к текстовому формату (например, JSON). Сама база (точнее, таблица с отпечатками) состоит всего из двух полей: `id` — уникального идентификатора пользователя, и `data` — отпечатка в текстовом формате.

3.4.5 Функция анализа

Методику анализа и вычисляемые параметры мы подробно разберём в п. 3.5, а здесь опишем алгоритмические особенности самой функции анализа.

Итак, эта функция преобразует регистр состояния в массив параметров. Параметры делятся по типам событий, и за каждый тип событий отвечает своя подзадача (например, параметры нажатий клавиш вычисляются отдельно от параметров прокрутки страницы). Поскольку массив событий общий, каждая функция просто пропускает «чужие» события. Если на очередной итерации поле `id` отличается больше чем на единицу, то пропущенные пакеты линейно интерполируются (поскольку в результате будут вычисляться, как правило, средние значения, точность от этого практически не пострадает).

Пример массива, вычисляемого функцией приведён на рис. 4.

```
{
    "keyboard": {
        "type_speed": 742,
        "keypress_length": 31.4159265,
        "has_more_than_10_fingers": 0,
    },
    "mouse": {
        "precision": 0.01011970,
        "cs_go_rank": 11
    },
    "gyro": {
        "speed_in_sea_knots": 2.71828183
    }
}
```

Рис. 4: Пример массива параметров (в формате JSON)

3.4.6 Сравнение параметров и построение вектора

Для построения вектора предположений нам нужно сравнить вычисленные параметры с такими же параметрами, сохранёнными в базе данных и представить разницу в виде числа от 0 до 1, где единица будет соответствовать полному точному соответствию всех параметров.

Как же сравнить два массива параметров? По сути, мы имеем дело с числовым вектором, и на пространстве таких векторов нетрудно ввести *меру*. Какие меры векторов вообще бывают? Самый простой вариант — сумма квадратов (кубов и т. д.) всех элементов вектора, иногда взятых с корнем соответствующей степени. Для сравнения же *двух* векторов можно взять их покоординатную разницу.

Но если просто сложить квадраты координат, при сравнении можно получить довольно странные результаты. Пусть, например, мы сравниваем два параметра клавиатуры — скорость набора в символах в минуту, и длительность нажатия клавиш в миллисекундах (назовём эти параметры `speed` и `length`).

Пусть вектор A равен `{"speed":600, "length":50}`, вектор B — `{"speed":550, "length":50}` и C = `{"speed":600, "length":90}`. Попробуем сравнить, к какому из векторов, B или C, находится ближе вектор A. В первом случае, мера разности векторов будет равна

$$\|A - B\| = (600 - 550)^2 + (50 - 50)^2 = 50^2 = 2500,$$

а во втором —

$$\|A - C\| = (600 - 600)^2 + (50 - 90)^2 = 40^2 = 1600.$$

Выходит, что вектор $(600, 50)$ гораздо ближе к вектору $(600, 90)$, чем к $(550, 50)$. Однако, интуиция подсказывает, что в первом случае одна из координат отличается почти вдвое, а во втором — менее, чем на 10%. Вывод очевиден — перед вычислением меры вектор следует *нормировать* — умножить каждую компоненту на соответствующий ей *коэффициент нормировки*.

Как вычислять эти коэффициенты? Один из вариантов — разделить значение параметра на некую *среднюю величину* этого параметра. Например, в нашем случае среднее значение параметра `speed` будет равно $\frac{600+600+550}{3} \approx 583.33$, а коээфициент для `length`, соответственно, $\frac{50+50+90}{3} \approx 63.33$. Таким образом, отнормированные векторы будут равны $A_n = (1.029, 0.79)$, $B_n = (0.943, 0.79)$ и $C_n = (1.029, 1.421)$, а результаты их сравнения составят $\|A_n - B_n\| = 0.0074$ и $\|A_n - C_n\| = 0.398$. Здесь совершенно очевидна более чем 50-кратная разница в пользу первого варианта.

Осталось сделать так, чтобы все числа укладывались в отрезок $[0, 1]$. Известно, что все наши меры будут заведомо больше нуля (так как являются суммой квадратов), нужно лишь разделить все значения на наибольшее из них. Мы получили набор чисел в промежутке от 0 до 1, однако, здесь точному совпадению соответствует значение 0, а не 1, поэтому вычтем все числа из единицы.

На примере наших векторов A, B и C, где, напомним, меры разности были равны $\|A_n - B_n\| = 0.074$ и $\|A_n - C_n\| = 0.398$, после нормировки результаты составят 0.814 и 0.000 соответственно. Это и будут те числа, которые мы запишем в поле `p` соответствующей записи нашего вектора.

У такого метода нормировки есть одна особенность, которую следует помнить — последний элемент вектора всегда будет в итоге равен нулю, даже если он всего один. Поэтому, когда в нашей базе только один пользователь, нужно предусмотреть иной механизм нормировки — например, заранее определить некий максимальный порог разности, и делить уже на него. В нашей модели применяется именно такой способ, но выбор методов нормировки, как и методов измерения вообще является вопросом открытым.

Для ещё более точного вычисления разницы можно применять дополнительный коэффициент для каждой компоненты — так называемый *вес*, или *стоимость* параметра. Эти коэффициенты можно подстраивать в процессе работы системы, используя самые различные методы — от простейших арифметических функций до многослойных нейронных сетей; впрочем, исследование этих методов выходит за рамки данной работы.

3.4.7 Подтверждение результатов и обучение системы

Когда вектор предположений вычислен, мы можем столкнуться с несколькими возможными вариантами интерпретации результатов:

- Пользователь определён однозначно, нет сомнений в верности результата;
- Пользователь выявлен, но лишь с небольшим отрывом по вероятности;

- Выявлено несколько пользователей с пренебрежимо малым различием;
- Ни один пользователь не подходит в достаточной мере.

Различать эти случаи предлагается с помощью установки пороговых значений и коэффициентов: например, если результат первого пользователя более чем вдвое превосходит результат второго, то пользователя можно считать определённым однозначно и вне всяких сомнений; если отрыв находится в пределах 30–100%, то пользователя можно считать идентифицированным, но «неуверенно»; наконец, если разрыв составляет менее 30 процентов, то идентификацию можно считать неудавшейся. Дополнительно к этому можно ввести минимальный порог для коэффициента — скажем, пользователя можно считать найденным, только если его «вероятность» составляет более 0.5.

Дальнейшие действия системы полностью зависят от случая, с которым мы столкнулись. Очевидно, что, если зашедший к нам пользователь в базе не нашёлся, то для него следует завести новую запись и сохранить в неё текущие параметры. Если же пользователь нашёлся, то API сервера должен вернуть его идентификатор. Но сделали ли мы всё, что нужно?

Поясним на примере всё с той же скоростью печати на клавиаутре. Предположим, к нам зашёл человек, который набирает 50 символов в минуту. Он в Сети лишь недавно, и скорость его невысока. Но вдруг он решил начать обучаться слепому набору на клавиатуре. В следующий раз он зайдёт на сайт, уже набирая, скажем, 70 симолов, затем 100, и через некоторое время этот же человек будет уже набирать все 400 символов в минуту. Сможет ли наша система его опознать?

Если никак не менять уже сохранённые данные, то, разумеется, не сможет — в базе-то записано, что человек набирает 50 символов и не более того. Отсюда следует очевидный вывод — сохранённый отпечаток пользователя нужно *подстраивать* всякий раз, когда он посещает наш сайт.

Простейший способ организации такого «обучения» системы состоит в том, чтобы просто перезаписывать данные в базе новым отпечатком пользователя. Именно он, в частности, используется в тестовой модели. Этот метод можно немного дополнить — например, полностью перезаписывать данные при «уверенной» идентификации, и сохранять среднее арифметическое между старыми и новыми данными — при «неуверенной». Можно пойти ещё дальше и устанавливать для старых и новых данных *веса*, с которыми они будут входить в сохранённый результат, а веса эти вычислять, например, на основе меры разности, отрыва от ближайшего «соперника» или ещё чего-нибудь.

Остался лишь случай, когда система выдаёт несколько пользователей с практически равными коэффициентами. В нашей модели этот случай считается равносильным полностью неудавшейся идентификации, т. е., в этом случае создаётся новая запись для пользователя; в некоторых случаях допускается неоднозначная идентификация клиента (например, если полученные данные передаются на вход

уже другой программе, которая далее отфильтрует заведомо неверные варианты), и система может выдавать всех пользователей сразу; как бы то ни было, тестовая система *всегда* выдаёт полный вектор предположений, а что делать дальше с этими данными — решает исследователь.

3.5 Параметры отпечатка и их анализ

Итак, мы полностью разобрали внутреннее техническое устройство нашей модельной системы. Осталось ответить на главный вопрос — каковы же будут те самые параметры, которые составят итоговый отпечаток?

После исследования эффективности и технической сложности анализа различных параметров системы, было решено остановиться на анализе трёх типов событий: набор текста на клавиатуре, прокрутка страницы и движения мыши.

3.5.1 Параметры клавиатуры

Первый параметр, который мы будем вычислять — *скорость печати*. Сделать это нетрудно — берём общее время работы и делим на число нажатий клавиш. Однако, тут есть один важный момент — время печати нужно брать *без пауз*. То есть, если человек напечатал, скажем, 5 слов, потом остановился, подумал и напечатал ещё 4, то время остановки не должно учитываться в подсчёте скорости. Как это учесть? Например, можно считать остановкой увеличение промежутка между нажатиями в N раз; в тестовой модели $N = 10$.

Ещё один интересный параметр, который можно посчитать — *длительность нажатия клавиш*. Что это такое? Формально, это время между нажатием клавиши и её отпусканием (обычно находится в районе 50–100 миллисекунд). В первую очередь, оно зависит от «ловкости» движений пользователя и косвенно от клавиатуры, за которой он работает. Учитывая, что наша система в основном предназначена для различения пользователей одной машины, различием клавиатур можно пренебречь.

Возьмём также *правильность печати*, или коэффициент опечаток. Как его считать? В простейшем случае, можно взять отношение числа нажатий клавиши Backspace к общему числу нажатий — для наших целей этого пока достаточно.

3.5.2 Прокрутка страницы

Будем рассматривать события прокрутки страницы отдельно от остальных событий мыши (хотя бы потому, что прокрутка может осуществляться не только мышью). И первый параметр, который будет нас интересовать — *скорость прокрутки*. Очевидно, что у всех людей немного разная скорость чтения текста, и этим можно воспользоваться. Правда, нужно снова учесть одну особенность: у человека, просматривающего сайт с экрана мобильного телефона абсолютная скорость прокрутки страницы будет в 5–10 раз меньше, чем у того, кто просматривает

его с широкоформатного монитора. Что с этим делать? Нужно умножить абсолютную скорость на коэффициент, пропорциональный ширине экрана (или, более точно, ширине текстового поля). В нашей модели используется ширина экрана, разделённая на 1000 — для сохранения удобочитаемости данных.

Помимо этого, скорость прокрутки, как и скорость клавиатуры, мы будем учитывать без пауз — вычтем все моменты времени, где пауза между событиями прокрутки многократно превысила средний временной промежуток.

Кроме скорости, будем учитывать ещё и *размах прокрутки* — среднее число «нормированных пикселей» (см. выше), на которое пользователь прокручивает страницу за один раз.

3.5.3 Движения мыши

Самая интересная и перспективная часть нашей системы фингерпринтинга. Здесь действительно есть где развернуться — число записываемых параметров может исчисляться десятками, при желании для идентификации можно применять сложные конструкции типа нейронных сетей; однако, в рамках данной работы мы ограничимся несколькими базовыми параметрами, которых, как будет показано далее, вполне достаточно для нашего исследования:

Скорость движения. Самый базовый параметр, для вычисления которого достаточно взять длину траектории и разделить на время. Как и в предыдущих случаях, из расчёта убираем все паузы;

Прямота движений. В случае отклонения мыши от прямой, будем сохранять величину угла отклонения. Просуммировав все такие углы и разделив на число «штрихов» (непрерывных движений мыши), мы получим среднее суммарное отклонение за одно движение;

Точность движений. Введём понятие точности как отношение длины пройденной траектории к расстоянию между начальной и конечной её точкой. Может показаться, что этот параметр напрямую связан с предыдущим, но на самом деле это не так — тракетория одной и той же длины может быть представлена ломаной как из двух звеньев, так и из десятков и сотен; разумеется, суммарное угловое отклонение будет различаться соответственно, при одинаковой длине траектории. Прямота движений характеризует такие особенности поведения человека, как плавность движений пальцев, дрожание руки, давление пальца на мышь или тачпад. Точность движений при этом определяется, в первую очередь, натренированностью пользователя и точностью его глазомера. Параметр берём, аналогично, в среднем за одно движение мыши;

Средний размах движений. Среднее перемещение (или путь — эти параметры связаны коэффициентом точности) мыши за одно движение. Чем больше это число, тем плавнее движения; напротив, чем оно меньше, тем движения более резкие и угловатые;

Средняя длительность клика. Тот же самый параметр, что и в анализе набора на клавиатуре, но обрабатывающий нажатия клавиш мыши.

Подытожим вышесказанное и приведём структуру готового массива (рис. 5).

```
{
    "keyboard":{
        "speed": 768, // скорость печати
        "length": 76.34, // длительность нажатий
        "typo": 0.054 // коэффициент опечаток
    },
    "scroll":{
        "speed": 0.418, // скорость прокрутки
        "span": 347 // размах прокрутки
    },
    "mouse":{
        "speed": , // скорость движения
        "straightness": 23.187, // прямота движений
        "precision": 1.94, // точность движений
        "span": , // размах движений
        "press_length": 77.13 // длительность нажатий
    }
}
```

Рис. 5: Пример готового массива парамтеров (в формате JSON)

3.6 Защита от проникновения

В силу широкой распространённости самых разных техник фингерпринтинга, не могли не появиться и средства, направленные на противодействие им. Сейчас сущетвует множество различных дополнений для браузеров (такие как AdBlock, Ghostery, Disconnect и др.), которые либо блокируют работу таких «жучков», либо намеренно «портят» отправляемые данные с целью сбить систему с толку и выдать себя за кого-то другого. Полную блокировку систем отслеживания, жучков и даже рекламы отследить в целом нетрудно (на многих сайтах такая защита уже давно установлена — они просто «не отдадут» страницу, пока клиент не выключит блокировщик рекламы), а вот с намеренной порчей данных несколько сложнее.

Как правило, это не отслеживается вообще никак, что позволяет в определённых случаях легко подделать отпечаток и не просто заслать «мусорные» данные, а выдать себя за вполне конкрентное другое устройство (например, поле UserAgent можно задать просто через настройки браузера). Однако, наша система спроектирована так, чтобы принципиально не допускать подобных попыток.

Поскольку все вычисления происходят на сервере, а клиент лишь порождает события и отправляет их, выдать себя за другого человека можно лишь полностью смоделировав исходный процесс его поведения на странице. Даже полностью зная отпечаток человека, смоделировать этот процесс довольно непросто (особенно, если добавить простейшие проверки «на человечность» со стороны сервера).

4 Тестирование модели

Итак, модель построена. Но наша задача — не просто построить модель, а *проверить гипотезу* о её применимости.

Для начала, точно сформулируем, что же мы, собственно, будем проверять. Гипотеза утверждала возможность построения системы, способной различать пользователей независимо от физической машины, за которой они находятся. Поиск пользователей происходит путём сличения их отпечатков с находящимися в базе данных — это значит, что сначала эти отпечатки следует собрать.

Таким образом, тестирование системы будет проходить в пять этапов:

1) Разработка тестового модуля для проведения эксперимента;

2) Поиск тестовой выборки — людей, готовых поучаствовать в эксперименте;

3) Первичный сбор отпечатков с участников эксперимента;

4) Проверка работы системы — повторное прохождение тестирования участниками и сбор обратной связи с её последующим анализом;

5) Подведение итогов тестирования и объявление результатов эксперимента.

4.1 Разработка тестового модуля

Разработанная система ввиду специфики своей работы лучше всего подходит для интеграции в различные *веб-страницы*. Поэтому, для проведения эксперимента мы будем создавать именно веб-страницу.

Наша тестировочная страница должна обладать рядом особых свойств — она должна быть построена специально для эксперимента с учётом его особенностей.

Поскольку использование системы требует активного использования всех средств ввода — клавиатуры, мыши, прокрутки, страница должна обязательно включать в себя следующие элементы:

а) Для проверки клавиатуры — поля для ввода текста;

б) Для прокрутки страницы — достаточный объём текстовой информации, чтобы его пришлось прокручивать даже на большом экране (но, в то же время, не слишком большой, чтобы не отпугнуть испытуемых);

в) Для движений мыши — разлчиные кнопки, флажки, переключатели и т. д.

Можно заметить, что на роль «тестового полигона» под эти параметры отлично подходит категория онлайн-тестов.

Итак, для проведения эксперимента было разработано два несложных теста: первый — для первичного сбора отпечатков, второй — для проверки работы нашей системы. Помимо самих тестов и подключения к ним клиентского скрипта, в тесты второго этапа был интегрирован специальный интерфейс *обратной связи*: в процессе прохождения теста, если испытуемый был успешно найден в базе, ему выдаётся всплывающее окно с предположением и две кнопки: для подтверждения и опровержения выданного результата (рис. 6); эти данные вместе с отпечатками обоих шагов также записываются в базу.

Помимо этого, для тестирования способностей системы по обнаружению отсутствия пользователей, несколько человек приняли участие только во втором этапе тестирования.

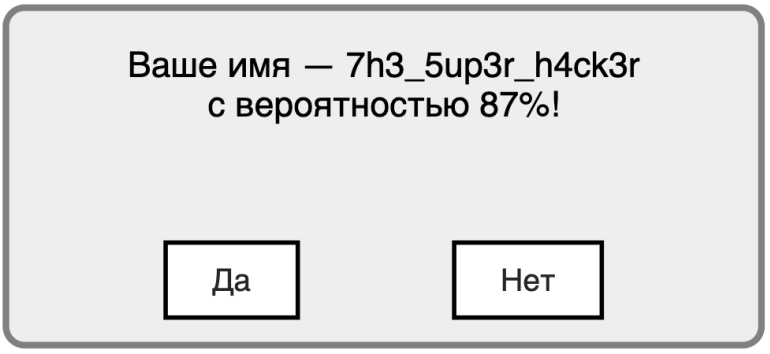

Рис. 6: Пример окна с подтверждением

4.2 Поиск выборки и проведение эксперимента

Кого нужно взять в качестве испытуемых, чтобы выборка адекватно отражала способности нашей системы?

В первую очередь, это должна быть достаточная выборка — для наших целей вполне хватит порядка 10 человек; во-вторых, это должны быть как люди из разных социальных групп (программисты, обычные пользователи, люди, далёкие от компьютеров), так и представители одного сообщества (например, можно ли различать пользователей среди программистов, многие из которых обладают одинаково высокой скорстью печати и точностью движений мыши?)

Тестирование было, как водится, проведено среди множества друзей и знакомых автора — всего 9 человек. Данные оказались крайне разношёрстными (полная выборка приведена в Приложении А).

5 Итоги работы

Начнём с количественных результатов эксперимента:

- Средняя «вероятность» идентификации пользователей составила 75%, минимальная — 51%, максимальная — 98%;
- Доля правильно идентифицированных пользователей — 89%; среди них средняя «вероятность» составила 95%, минимальная — 92%, максимальная — 98%;
- Доля неправильно идентифицированных пользователей составила 11%, всего 1 человек;

Вышеуказанные итоги показывают, что даже такая простая модель идентификации в целом способна с достаточной точностью определять пользователей.

6 Перспективы проекта

Эксперимент данной работы оказался довольно успешным, а поэтому встаёт вопрос о реальных сферах применения результатов исследования: где могут стать полезными данные методики идентификации?

6.1 Сферы реального применения

В целом, понятно, что мало кому придёт в голову пытаться идентифицировать таким образом миллиарды компьютеров, но это и не нужно — для отслеживания физических устройств существуют гораздо более эффективные методы; вместо этого, данную систему было бы удобно применять там, где обычный фингерпринтинг оказывается бессильным: в первую очередь, это различение нескольких пользователей одной и той же машины. Сценариев использования для такой системы можно придумать великое множество, вот лишь несколько примеров:

- Различение пользователей одного устройства для правильного таргетирования рекламы, результатов поиска и т. п. (см. п. 2.1);
- Защита активных аккаунтов, открытых в браузере, от постороннего доступа. Классическая ситуация: ребёнок сидит в своём аккаунте в социальной сети и ведёт личные переписки; вдруг, родитель просит ребёнка срочно передать ему компьютер, например, чтобы сделать что-то по работе; после чего родитель случайно (а иногда совсем даже и не случайно) обнаруживает открытый аккаунт ребёнка со всеми перепсиками, последствия чего могут оказаться довольно непредсказуемыми и всецело зависящими от адекватности родителя;

- Дополнительная защита аутентификации: помимо пары логин-пароль, в базу также записывается отпечаток, после чего, в случае значительного расхождения полученного отпечатка с базой, включается «подозрительный режим» и пользователю, например, предлагается подтвердить вход с помощью мобильного телефона или электронной почты;

6.2 Дальнейшее развитие

При грамотном подходе проект имеет большие перспективы; однако, тестовая модель пригодна лишь для исследовательских целей, и до финальной версии должна пройти множество доработок. Вот лишь некоторые из них:

- Значительное расширение круга записываемых параметров (в том числе, возможно, с новых типов устройств);
- Совершенствование методик сравнения (обучаемые нормировочные коэффициенты, другие способы измерения и пр.);
- Совершенствование методик обучения (обучаемые весовые коэффициенты, другие способы обучения и пр.);
- Уточнение порогов распознавания (в т. ч., в динамическом режиме);
- Упрощение и расширение возможных сфер внедрения, оптимизация кода, разработка дружественного интерфейса и т. д.

7 Заключение

Фингерпринтинг давно и прочно вошёл в обыденную жизнь Интернета. Как правило, он используется для показа таргетированной рекламы, и, несмотря на общую нелюбовь к её объёму, это всё ещё не самое «плохое», для чего может использоваться фингерпритинг. Помимо маркетинговых целей, фингерпринтинг может также использоваться для не самой честной и открытой слежки за людьми, причём даже не со стороны спецслужб (это хотя бы можно объяснить необходимостью обеспечения безопасности), а рядовыми коммерческими компаниями. Что потом произойдёт с этими данными — известно лишь самим этим компаниям.

Данная работа является одной из немногих попыток использовать фингерпринтинг в «мирных» целях — для *защиты* конфиденциальности, а не уничтожения в самом её корне. Как бы то ни было, фингерпринтинг — лишь технология, инструмент, а как именно тот или иной человек будет этот инструмент использовать — целиком на его совести.

A Список отпечатков тестовой выборки

Полная база находится в файле `Base.json`.

Ввиду её большого объёма, приведём в тексте работы её фрагмент:


```
"Пётр":{
        "keyboard":{
            "speed":203,
            "length":104.463157894736838215976604260504 25,
            "typo":0.0666666666666666657414808128123695431640625
        },
        "scroll":{
            "speed":0.00400368966145502654802035280567905169677734375,
            "slice_span":227.610871794871769679957651533906 25,
            "slice_time":274.60256410256408798886695876762 5,
            "slice_count":78
        },
        "mouse":{
            "speed":0.246787569288549918367081659198447594921875,
            "precision":1.41076854869755319299429174861808593 75,
            "span":919.3023417717503207313711754977703 09
            "stroke_time":3725.0755555555556384206283837125,
            "press_length":1449.49230769230780424550175 6,
            "stroke_count":225
        }
},
"Kostya":{
    "keyboard":{
        "speed":204,
        "length":128.32432432432432278801570646464824 6875,
        "typo":0.08791208791208791895055441045769839541 1083984375
    },
    "scroll":{
        "speed":0.040158429784103222281999023834941908654296875,
        "slice_span":264.55655555555551927682245150208640625,
        "slice_time":366.16666666666685614472953602678046875,
        "slice_count":18
    },
    "mouse":{
        "speed":0.913167506067501855149259881727630272150390625,
        "precision":1.48403953676147160045672990236198710693359375,
        "span":748.56906309883459016418782994151115417,
        ...
```

В Исходные коды программного продукта

Коды клиентской и серверной частей системы находятся в файлах `main.php`, `test.php`, `client.js`, `test-client.php`.

Также исходные коды могут быть загружены по адресу `github.com/himotokun/teh_fingerprint`.

В.1 Инструкции по подключению системы

Для подключения потребуется любой сервер с поддержкой PHP. Система состоит из двух частей: клиента и сервера. Сервер, в свою очередь, состоит из *ядра* (`main.php`) и *интерфейса* (API, `test.php`). Клиентская часть представлена, главным образом, скриптом `client.js`. Этот скрипт необходимо вставить в тело страницы, на которой будет происходить идентификация.

Результаты идентификации сохраняются в базу MySQL. Параметры базы задаются в методе `tfpServer::loadSqlTable()`. База состоит из трёх полей — числовое поле `id` (первичный ключ, автоинкремент) и текстовые поля `nickname` и `data`. Поле `nickname` не является обязательным, но используется в тестовой модели в элементах пользовательского интерфейса.

Для тестирования работы системы был создан файл `test-client.php`. Для работы с ним требуется включить его в `.php`-файл нужной страницы и после основной части страницы вызвать функцию `loadFingerprint($mode)`, где `$mode` равен `'store'` для принудительного сохранения собранной информации под новым именем (используется в первой части тестового модуля), либо `'check'` для анализа получаемой информации и выдачи диагностического окна с результатами вычислений (рис. 6).

Справочно схема подключения представлена на рис. 7.

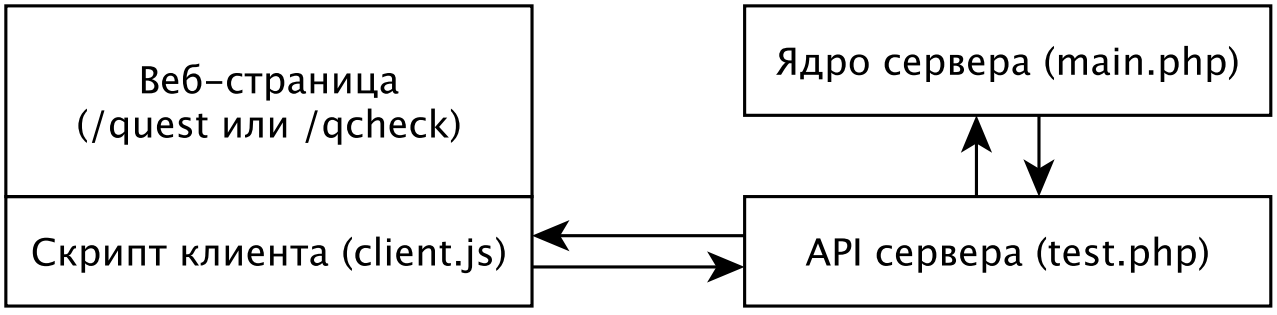

Рис. 7: Диаграмма работы тестовой системы

Список литературы

[1] Monaro M., Gamberini L., Sartori G. *The detection of faked identity using unexpected questions and mouse dynamics.* PLoS ONE 12(5): e0177851, 2017.

https://doi.org/10.1371/journal.pone.0177851

[2] Jose Carlos Norte. *Advanced Tor Browser Fingerprinting*, 2016.

http://jcarlosnorte.com/security/2016/03/06/advanced-tor-browser-fingerprinting.html

[3] Jay R. Young, Randall S. Davies, Jeffrey L. Jenkins & Isaac Pfleger. *Keystroke Dynamics: Establishing Keyprints to Verify Users in Online Courses.* Computers in theSchools, 2019.

https://doi.org/10.1080/07380569.2019.1565905

[4] Hibbeln, Martin & Jenkins, Jeffrey & Schneider, Christoph & Valacich, Joseph & Weinmann, Markus. *How Is Your User Feeling? Inferring Emotion Through Human-Computer Interaction Devices.* MIS Quarterly. 41. 10.25300/MISQ/2017/41.1.01, 2017.