Данная работа содержит попытки практической реализации идентификации аппаратной части ПК на основе матриц скорости выполнения кода на основе работы «Clock Around the Clock: Time-Based Device Fingerprinting» от Iskander Sanchez-Rola

Оглавление

Обоснование и реализация динамической идентификации аппаратной части ПК на основе матрицы скорости выполнения кода.....2

Первая попытка реализации динамической идентификации.....3

Уточнение ключевых деталей.....6

Вторая попытка реализации динамической идентификации.....9

Стабилизация результатов динамической идентификации.....21

Рекомендации по дальнейшему исследованию.....27

Список используемых источников.....28

Обоснование и реализация динамической идентификации аппаратной части ПК на основе матрицы скорости выполнения кода

Динамическая идентификация аппаратной части ПК - альтернативный метод идентификации ПК на основе измерения некоторых уникальных параметров, совокупность которых может однозначно идентифицировать множество ПК с одинаковыми аппаратным и программным обеспечением. Такую генерацию HWID сложнее обнаружить и обойти, т.к. отсутствует явная ссылка на какой-либо аппаратный атрибут.

В работе «Clock Around the Clock: Time-Based Device Fingerprinting» от 2018 года группа исследователей под руководством Искандера Санчерз-Рола продемонстрировала и описала метод, основанный на множественном измерении скорости выполнения кода [7]. В статье утверждается, что сгенерированные измерения (около 50 тысяч вызовов одной и той же функции) будут уникальны для каждого ПК за счёт различий в кристаллах кварца из-за особенности производства. Внутренние часы ПК используют генераторы на основе кристаллов кварца, и небольшие колебания в этих кристаллах могут привести к чрезвычайно малым, но измеримым различиям в тактовой частоте. На измерение скорости выполнения кода могут повлиять многие факторы:

1. Пропуски кэша/TLB и совместное использование ресурсов конвейера другими потокам процессора.
2. Использование Hyper-threading
3. Применение динамического масштабирования частоты и напряжения процессора (DVFS)

Из-за данных неопределённых факторов одного измерения недостаточно для получения стабильного результата скорости выполнения.

После генерации матрицы скорости выполнения одной и той же функции с различными входными параметрами, она сравнивается с другой матрицей и определяется, принадлежат ли эти матрицы к одному и тому же ПК или нет. Авторы

реализовали нативную версию для исполнения на ПК (язык программирования C) и версию для исполнения в браузере (язык программирования JavaScript), обе версии имеют название CryptoFP т.к. измеряют скорость выполнения функции CryptGenRandom().

Помимо динамического измерения времени выполнения кода, отличительной чертой данного подхода является отсутствие строго сравнения двух идентификаторов на предмет сходства. Вместо этого, вычисляется порог идентичности двух матриц и, если он превышает 50%, то обе матрицы принадлежат к одному и тому же ПК.

Первая попытка реализации динамической идентификации

Был реализован прототип для тестирования описанного механизма на языке программирования Delphi. На рисунке 11 изображен код генерации матрицы скорости выполнения функции RandBytes(). Внутри функции RandBytes() вызывается CryptGenRandom(), заполняющая буфер криптографически случайными байтами.

```
procedure RandBytes(const ALen: Integer);
begin
  if not CryptGenRandom(Prov, ALen, @BufForRndVals[0]) then
    raise Exception.CreateFmt('CryptGenRandom failed! GLE = ',
      [GetLastError()]);
end;

function FPGeneration(const n, M: Integer): TTimeMatrix;
var
  iCounterPerSec: Int64;
  I, J: Integer;
  T1, T2, DiffT: Int64;
  Diff: Double;
begin
  I := 0;
  SetLength(Result, M, n);
  QueryPerformanceFrequency(iCounterPerSec);

  while I < M do
  begin
    J := 0;

    // Повторение вызова с разными параметрами указанное кол-во раз
    while J < n do
    begin
      QueryPerformanceCounter(T1);

      // Получение случайного кол-ва байт
      // Минималка - 2 байта
      RandBytes(J + 2);

      QueryPerformanceCounter(T2);
      // Считаем сколько прошло тактов между двумя точкам
      DiffT := (T2 - T1);
      Diff :=  (DiffT * 1000000) / iCounterPerSec;
      // Получение миллисекунд - * 1000
      // Получение микросекунд - * 1000000
      Result[I][J] := DiffT;

      J := J + 1;
    end;
    I := I + 1;
  end;
end;
```

Рисунок 11 – Алгоритм генерации матрицы времени выполнения

После этапа генерации были реализованы методы получения mode каждой строки матрицы (нахождение наиболее повторяющихся значений в строке матрицы для избавления от помех и переключений планировщика ОС), что изображено на рисунке 12. Так же на рисунке 12 представлен алгоритм определения идентичности матриц на основе подсчёта суммы вхождений mode строк двух матриц и применения порога в 50%.

```
function FPCheck(const fp1, fp2: TTimeMatrix; const n, M: Integer;
  Trashold: Single): boolean;
var
  NumCon1, NumCon2, SumCon: Integer;
  ResPercentage: Single;
begin
  NumCon1 := GetNumCoincidences(fp1, fp2, n, M);
  NumCon2 := GetNumCoincidences(fp2, fp1, n, M);
  SumCon := NumCon1 + NumCon2;

  WriteLn(Format('FPCheck: NumCon1 = %d, NumCon2 = %d, SumCon = %d',
    [NumCon1, NumCon2, SumCon]));

  ResPercentage := SumCon / (n * 2) * 100;

  WriteLn(Format('FPCheck: ResPercentage = %f Trashold = %f',
    [ResPercentage, Trashold]));

  Result := ResPercentage >= Trashold;
end;
```

```
function GetNumCoincidences(const fp1, fp2: TTimeMatrix;
  const n, M: Integer): Integer;
var
  num_coindences: Integer;
  I, J: Integer;
  fp1_mode: TArray<Double>;
  check: boolean;
begin
  num_coindences := 0;

  SetLength(fp1_mode, Length(fp1));

  // Находим mode в каждой строке
  for I := low(fp1) to high(fp1) do
  begin
    fp1_mode[I] := ComputeMode(TArray<Int64>(fp1[I]));
  end;

  for I := 0 to n - 1 do
  begin
    check := FALSE;
    J := 0;

    while (J < M) and (not check) do
    begin
      if fp1_mode[I] = fp2[I][J] then
      begin
        Inc(num_coindences);
        check := True;
      end
      else
        Inc(J);
    end;
  end;

  Result := num_coindences;
end;
```

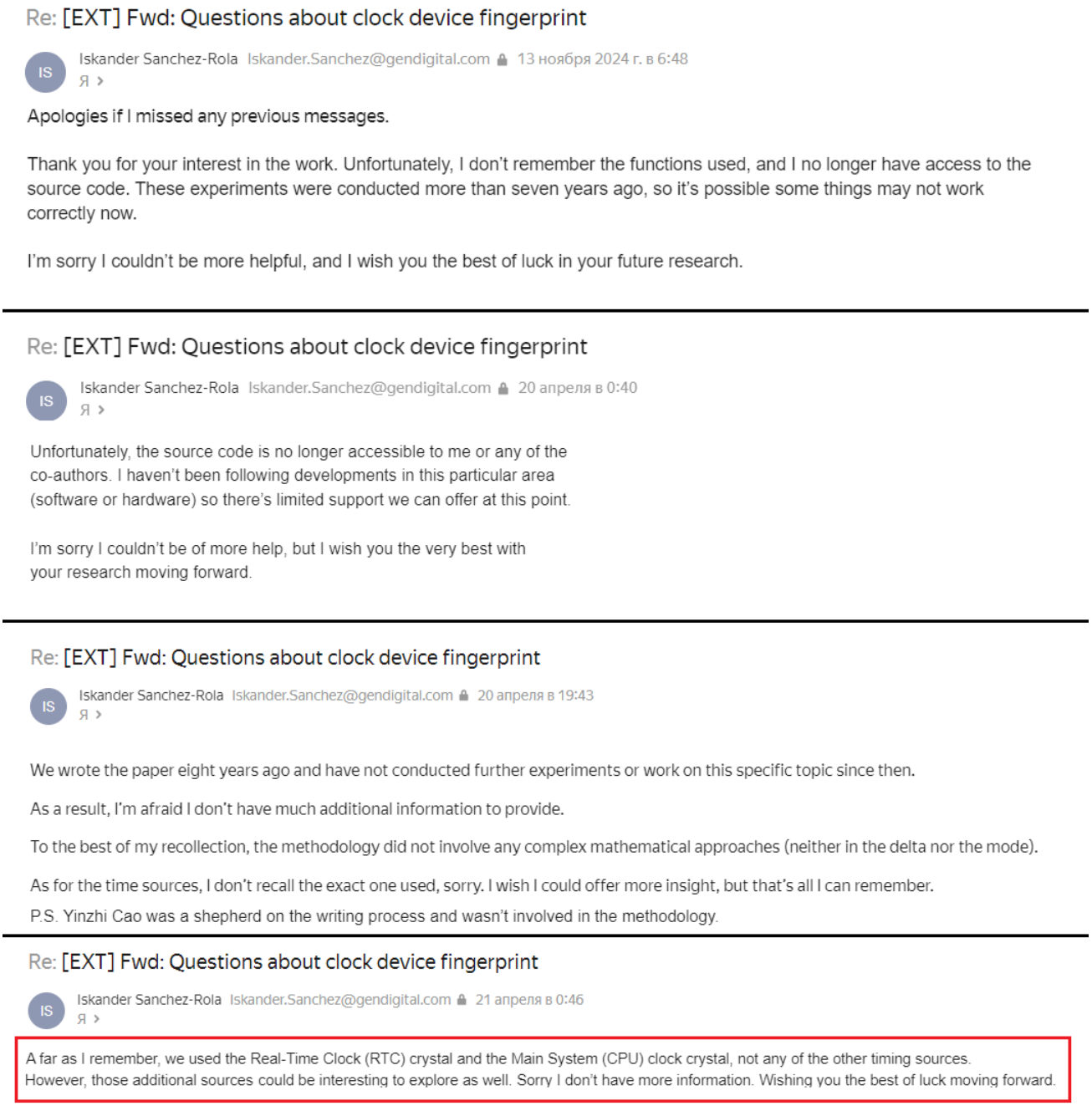
```
function FindMode(const A: TArray<Double>): Double;
var
  Counts: TDictionary<Double, Integer>;
  MaxCount: Integer;
  Mode: Double;
  I: Integer;
begin
  // GPT сгенерировал код намного короче
  Counts := TDictionary<Double, Integer>.Create;
  try
    // Подсчитываем количество вхождений каждого элемента
    for I := Low(A) to High(A) do
    begin
      if not Counts.ContainsKey(A[I]) then
        Counts.Add(A[I], 0);

      Counts[A[I]] := Counts[A[I]] + 1;
    end;

    // Находим элемент с наибольшим количеством вхождений
    MaxCount := 0;
    Mode := 0;
    for I := Low(A) to High(A) do
    begin
      if Counts[A[I]] > MaxCount then
      begin
        MaxCount := Counts[A[I]];
        Mode := A[I];
      end;
    end;

    Result := Mode;
  finally
    Counts.Free;
  end;
end;
```

Рисунок 12 – Алгоритм поиска рассчитанного mode матрицы fp1 в матрице fp2 и функция поиска mode строки матрицы

В ходе разработки прототипа тестировалось несколько вариантов замера времени, однако добиться однозначной работоспособности не удалось. Было выдвинуто предположение, что в статье Искандера отсутствует часть ключевой информации, необходимой для реализации рабочего варианта CryptoFP.

Уточнение ключевых деталей

После неудачных попыток воспроизвести стабильный результат генерации матрицы выполнения были разосланы письма авторам данной статьи для уточнения ключевых деталей. На рисунке 13 изображены ответы Искандера на вопросы касаемо реализации нативной версии генерации матрицы скорости выполнения кода – CryptoFP.



Re: [EXT] Fwd: Questions about clock device fingerprint

IS Iskander Sanchez-Rola Iskander.Sanchez@gendigital.com 13 ноября 2024 г. в 6:48
Я

Apologies if I missed any previous messages.

Thank you for your interest in the work. Unfortunately, I don't remember the functions used, and I no longer have access to the source code. These experiments were conducted more than seven years ago, so it's possible some things may not work correctly now.

I'm sorry I couldn't be more helpful, and I wish you the best of luck in your future research.

Re: [EXT] Fwd: Questions about clock device fingerprint

IS Iskander Sanchez-Rola Iskander.Sanchez@gendigital.com 20 апреля в 0:40
Я

Unfortunately, the source code is no longer accessible to me or any of the co-authors. I haven't been following developments in this particular area (software or hardware) so there's limited support we can offer at this point.

I'm sorry I couldn't be of more help, but I wish you the very best with your research moving forward.

Re: [EXT] Fwd: Questions about clock device fingerprint

IS Iskander Sanchez-Rola Iskander.Sanchez@gendigital.com 20 апреля в 19:43
Я

We wrote the paper eight years ago and have not conducted further experiments or work on this specific topic since then.

As a result, I'm afraid I don't have much additional information to provide.

To the best of my recollection, the methodology did not involve any complex mathematical approaches (neither in the delta nor the mode).

As for the time sources, I don't recall the exact one used, sorry. I wish I could offer more insight, but that's all I can remember.

P.S. Yinzhi Cao was a shepherd on the writing process and wasn't involved in the methodology.

Re: [EXT] Fwd: Questions about clock device fingerprint

IS Iskander Sanchez-Rola Iskander.Sanchez@gendigital.com 21 апреля в 0:46
Я

A far as I remember, we used the Real-Time Clock (RTC) crystal and the Main System (CPU) clock crystal, not any of the other timing sources. However, those additional sources could be interesting to explore as well. Sorry I don't have more information. Wishing you the best of luck moving forward.

Рисунок 13 – Ответы Искандера Санчез-Ролы по email на вопросы реализации CryptoFP

Из ответов Искандера стало понятно, что доступ к исходному коду нативной версии приложения CryptoFP был утерян, а ключевые детали позабыты. Однако было получено подтверждение что для замера времени выполнения кода использовалось стандартное Windows API для работы с датой и временем, а так же была исключена работа с таймерами GPU.

Был так же отправлен запрос в университет Деусто в кампус Бильбао в котором автор статьи в 2018 году проводил эксперименты на большом количестве ПК с одинаковыми аппаратными составляющими. На рисунке 14 изображён ответ от кампуса Бильбао с обещанием помочь с получением информации о характеристиках ПК, на которых происходило тестирование нативной версии CryptoFP в 2018 году, однако дальнейших коммуникаций не последовало.

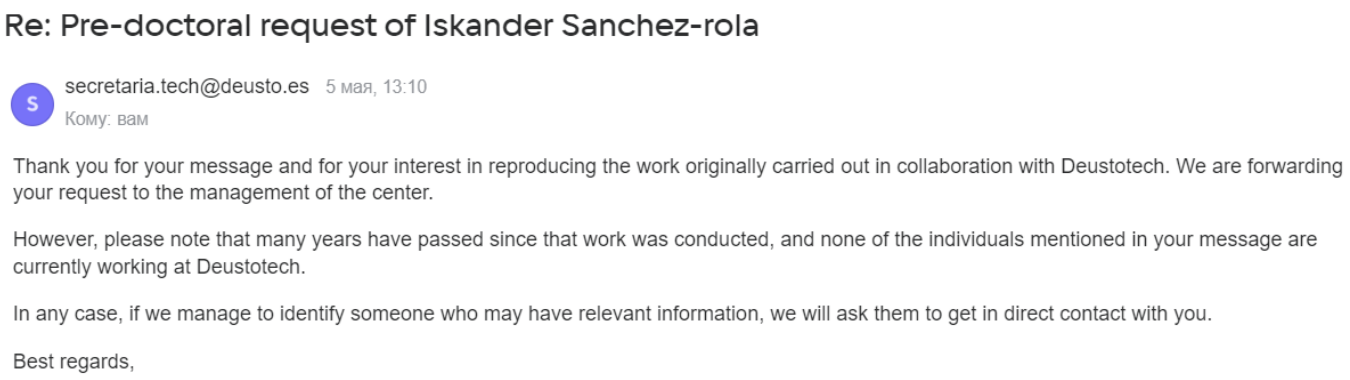
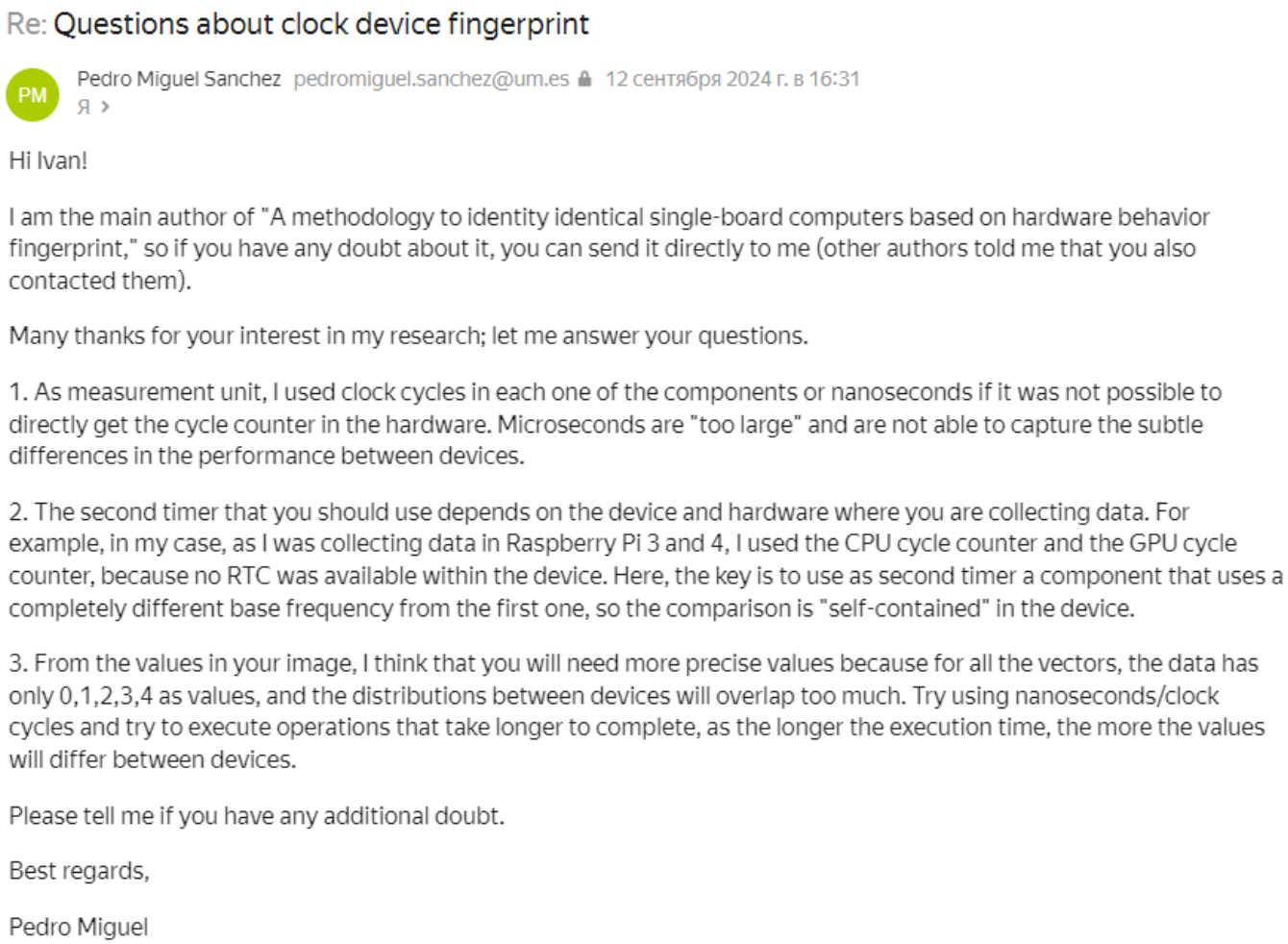


Рисунок 14 – Ответ университета Деусто кампуса Бильбао по email на запрос информации о характеристиках ПК и исходном коде CryptoFP

В ходе анализа других научных работ, ссылающихся на статью Искандера, удалось связаться с автором исследования от 2023 года «A methodology to identity identical single-board computers based on hardware behavior fingerprint» [8] Педро Мигелем Санчезом, который так же пытался воспроизвести механизм идентификации устройств на основе генерации матрицы скорости выполнения кода. На рисунке 15 изображён ответ Педро Мигеля Санчеза на вопросы реализации CryptoFP.



Re: Questions about clock device fingerprint

PM Pedro Miguel Sanchez pedromiguel.sanchez@um.es 12 сентября 2024 г. в 16:31
Я >

Hi Ivan!

I am the main author of "A methodology to identity identical single-board computers based on hardware behavior fingerprint," so if you have any doubt about it, you can send it directly to me (other authors told me that you also contacted them).

Many thanks for your interest in my research; let me answer your questions.

1. As measurement unit, I used clock cycles in each one of the components or nanoseconds if it was not possible to directly get the cycle counter in the hardware. Microseconds are "too large" and are not able to capture the subtle differences in the performance between devices.

2. The second timer that you should use depends on the device and hardware where you are collecting data. For example, in my case, as I was collecting data in Raspberry Pi 3 and 4, I used the CPU cycle counter and the GPU cycle counter, because no RTC was available within the device. Here, the key is to use as second timer a component that uses a completely different base frequency from the first one, so the comparison is "self-contained" in the device.

3. From the values in your image, I think that you will need more precise values because for all the vectors, the data has only 0,1,2,3,4 as values, and the distributions between devices will overlap too much. Try using nanoseconds/clock cycles and try to execute operations that take longer to complete, as the longer the execution time, the more the values will differ between devices.

Please tell me if you have any additional doubt.

Best regards,

Pedro Miguel

Рисунок 15 – Ответ Педро Мигеля Санечза по email на вопросы реализации генерации матрицы выполнения кода CryptoFP

Педро Мигель Санчез сообщил, что для работоспособности механизма необходимо сравнивать результаты выполнения кода двух различных устройств с кварцевыми кристаллами, при этом их точность должна быть крайне высокой (необходимо сравнивать количество тактов выполнения кода, а не миллисекунды и микросекунды).

В статье Педро Мигеля Санчеза механизм Искандера показал свою неработоспособность из-за использования специфичного оборудования (Raspberry Pi 4 Model B и Raspberry Pi 3 Model B) в котором отсутствует второй точный таймер (RTC), необходимый для генерации и сравнения матриц [8].

Собрав ответы авторов воедино, были сделаны следующие выводы:

1. Для замеров скорости выполнения кода использовалось стандартные Windows API – Искандер.
2. Взаимодействия с таймерами GPU посредством CUDA/OpenCL не происходило – Искандер.
3. Необходимо использовать минимум два разных таймера с разной частотой. Одного таймера недостаточно для стабильного результата. Чем сильнее частота таймеров будет отличаться друг от друга, тем лучше – Мигель.
4. Чем точнее будут оба таймера, тем лучше. Рекомендуется работать с наносекундами или тиками процессора для замера скорости выполнения кода – Мигель.
5. Использовалось RTC и таймер процессора/системный таймер – Искандер. Автор до конца не помнит деталей и путается в понятиях таймеров.
6. Отсутствуют ключевые данные о характеристиках ПК, на которых в 2018 году Искандер производил тестирование стабильности нативной версии CryptoFP. В статье упоминается что тестирование происходило на 100-200 одинаковых ПК с ОС Windows 7 и Linux, но никакой информации о процессорах, количестве оперативной памяти, материнских платах и видеокартах этих ПК в статье нет.
7. Для каждого таймера необходимо генерировать свою матрицу, после чего следует вычесть одну матрицу из другой по модулю. Разница двух матриц будет являться матрицей устройства – Мигель.

Вторая попытка реализации динамической идентификации

С учётом ответов авторов научных статей было принято решение изучить и протестировать способ измерения скорости выполнения кода на основе тиков процессора как наиболее точный вариант измерений.

Все дальнейшие тесты происходили на двух ПК, характеристики которых представлены в таблице 7. Язык разработки тестового консольного приложения – Delphi 12. Компиляция для тестов выполнялась сперва под x32, а затем и под x64.

Таблица 7 – Характеристики ПК для тестирования

Процессор	**Видеокарта**	**ОЗУ**	**Материнская плата**	**Тип диска, где располагалась программа**	**ОС**
Intel Core i5-3230M 2.6 ГГц	NVIDIA GT 640M 2 Гб	DDR3 6 Гб	Acer VA70_HC	SSD	Windows 7 7601 SP1
Intel Core i7-7700 3.6 ГГц	NVIDIA GTX 1060 6 Гб	DDR4 16 Гб	MSI B150 GAMING M3 (MS-7978)	SSD	Windows 10 22H2 19045.5737

Для стабилизации результатов тестирования скорости выполнения кода при запуске тестируемой программы выполнялись следующие шаги:

1. Вызывался метод WinAPI SetProcessAffinityMask() для ограничения исполнения кода всех потоков приложения на одном ядре процессора. Это необходимо чтобы во время длительного замера кода выполнения, процессор не перекидывал исполнение задачи на другое ядро, что может повлиять на итоговые результаты.
2. Вызывался метод WinAPI SetPriorityClass() для установки приоритета процесса в HIGH_PRIORITY_CLASS и исключения вытеснения кода потока приложения планировщиком потоков ОС.

После чего происходило измерение скорости выполнения кода посредством выполнения ассемблерной инструкций RDTSC. Инструкция RDTSC возвращает в регистрах EDX и EAX кол-во тиков с момента последнего сброса процессора. В современных системах счётчик RDTSC является инвариантным iTSC, то есть не

зависит от использования технологий энергосбережения и увеличивается с постоянной частотой. Псевдокод использования RDTSC изображён на рисунке 16.

```
LFENCE()
StartTicks = RDTSC()

;...измеряемый код...

LFENCE()
EndTicks = RDTSC()

DiffTicks = EndTicks - StartTicks
```

Рисунок 16 – Псевдокод использования ассемблерной инструкции RDTSC

Стоит учитывать, что для корректного использования RDTSC необходимо сбрасывать конвейер команд процессора, в противном случае процессор может переупорядочить выполнение команд и RDTSC будет выполнено после вызова измеряемого кода [14]. Конвейер процессора - это последовательность этапов, через которые проходит каждая команда до полного выполнения (декодирование, исполнение, сохранение результата и т.д.).

В данном случае применяется ассемблерная инструкция LFENCE, но так же допускается применение CPUID. Команда CPUID принудительно очищает конвейер, а LFENCE ожидает выполнения всех команд и только затем продолжает выполнение [14].

На рисунке 17 изображён полный ассемблерный листинг команд для замера скорости времени выполнения кода в тиках с использованием команды RDTSC. В StartProfileRDTSC.inc происходит работа со стеком, получается текущее значение счётчика тиков процессора, сохраняемое в стеке. После чего происходит выполнение измеряемой функции CryptGenRandom(). Inc-файлы удобны для хранения ассемблерных вставок, которые можно применять в любом месте программы, скомпилированной под x32.

StartProfileRDTSC.inc

```
asm
  push ebp        ;// Сохраняем базовый указатель кадра
  mov ebp, esp    ;// Устанавливаем новый кадр стека
  sub esp, 32     ;// Выделяем место для локальных переменных

  push ebx
  push ecx
  push edx

  lfence          ;// подождать исполнения уже загруженного кода
  rdtsc           ;// счётчик TSC на входе в чистый конвейер
  push eax        ;// Сперва младшее число в стек
  push edx        ;// Затем старшее
end;
```

EndProfileRDTSC.inc

```
asm
  lfence              ;// подождать очистки конвейра
  rdtsc               ;// счётчик на выходе

  pop ecx             ;// счётчик на входе EDX (старший)
  pop ebx             ;// счётик на входе EAX (младший)

  sub eax,ebx         ;// вычислить разницу!
  sbb edx,ecx         ;// EDX:EAX = кол-во тиков

  push edx              ;// значения для сопроца
  push eax

  finit                 ;//  сопроцессор (thk Marylin)
  fild    qword[esp]    ;// взять тики со-стека в регистр ST0
  fst     [profTick]    ;// сохранить их в переменной для вывода
  fidiv   [CpuFreqMHz]     ;// получить время = тики / частоту CPU
  fstp    [profTime]    ;// сохранить время в переменной!

  add     esp,8         ;// восстановить стек от пушей..

  pop edx
  pop ecx
  pop ebx

  add esp, 32     ;// Освобождаем выделенную память
  pop ebp         ;// Восстанавливаем старый кадр стека
end;
```

Рисунок 17 – Ассемблерный листинг команд для замера скорости выполнения кода

После выполнения функции CryptGenRandom() вызывается EndProfileRDTSC.inc для получения текущего значения счётчика тиков, подсчёта разницы между тиками и получения времени выполнения путём деления тиков на

частоту CPU в сопроцессоре. На рисунке 18 демонстрируется генерация матрицы скорости выполнения кода с сохранением тиков RDTSC.

```
function FPGenerationCPU(const n, M: Integer): TTimeMatrix;
var
  PrerfFreq: Int64;
  I, J: Integer;
  T1, T2, DiffT: Int64;
  Diff: Double;
begin
  I := 0;

  SetLength(Result, n, M);

  while I < n do
  begin
    J := 0;

    // Инициализация параметров ПЕРЕД началом каждой новой строк
    // что бы по одним и тем же параметрам располагались одни и те же значения
    InnerCounter := START_BYTE;
    CurParam := 0;

    // Повторение вызова с разными параметрами указанное кол-во раз
    while J < M do
    begin

{$I 'Asm/StartProfileRDTSC.inc'}
      // Получение случайного кол-ва байт
      wrapRandBytes();
{$I 'Asm/EndProfileRDTSC.inc'}

      Result[I][J] := profTick;

      J := J + 1;
    end;
    I := I + 1;
  end;

  Diff := 0;
end;

procedure wrapRandBytes;
const
  MAX_BUF_LEN = 10240;
begin
  InnerCounter := InnerCounter + STEP;

  if InnerCounter >= MAX_BUF_LEN then
    InnerCounter := START_BYTE;

  RandBytes(InnerCounter);
end;
```

Рисунок 18 – Генерация матрицы выполнения кода на основе RDTSC

Для получения частоты CPU в МГц используется замер времени выполнения функции WinAPI Sleep() с рагументом 500 мсек, что изображено на рисунке 19.

Получаемая частота является максимальной частотой на которой может работать процессор. В EndProfileRDTSC.inc сохраняется два значения – кол-во тиков и время в микросекундах, затраченное на выполнение функции CryptGenRandom().

```
function GetCpuFrequencyMHz: UInt32; register;
asm
  push ebp        ;// Сохраняем базовый указатель кадра
  mov ebp, esp    ;// Устанавливаем новый кадр стека
  sub esp, 8      ;// Выделяем место для локальных переменных

  push ebx        ;// сохраним в стеке
  push ecx
  push edx

  lfence
  rdtsc

  push     eax          ;// сохраняем только младшую часть TSC

  push 500              ;// msec
  call Sleep            ;//

  lfence
  rdtsc

  pop      ebx          ;//
  sub      eax,ebx      ;// TSC за 0,5 сек
  mov      ebx,500000   ;// перевести в MHz
  xor      edx,edx      ;// EDX:EAX сброс EDX чтобы не влияло на результат
  div      ebx          ;//

  pop edx               ;// Восстановим значения для вызывающего кода
  pop ecx
  pop ebx

  add esp, 8       ;// Освобождаем выделенную память
  pop ebp          ;// Восстанавливаем старый кадр стека
  ;// EAX = частота в MHz
end;
```

Рисунок 19 – Получение частоты CPU

Для более точных измерений был реализован метод подсчёта кол-ва погрешности тиков, затрачиваемых на команды, участвующие в процессе измерения, а именно, кол-во тиков которое занимает команда LFENCE и две команды PUSH [14], изображённое на рисунке 20. Однако, вычитание данных тиков в EndProfileRDTSC.inc не повлияло на точность измерений.

```
function GetFixTicks: UInt32; register;
// Узнаем сколько тиков занимает 2 PUSH + LFENCE чтобы не учитывать
// их при измерениях.
asm
  push ebp          ;// Сохраняем базовый указатель кадра
  mov ebp, esp      ;// Устанавливаем новый кадр стека
  sub esp, 8        ;// Выделяем место для локальных переменных

  push ebx          ;// сохраним в стеке
  push ecx
  push edx

  lfence
  rdtscp

  push      eax     ;// младшие 4 байта RDTSC, будем вычитать их
  push      edx     ;// старшие 4 байта RDTSC, мы с ними не работаем

  lfence
  rdtscp            ;// Получаем тики End.EAX:EDX

  pop       ebx         ;// Start.EDX, скипаем т.к. мелкая разница в тиках
  pop       ebx         ;// Start.EAX - работаем с ним
  sub       eax,ebx     ;// End.EAX - Start.EAX

  pop edx
  pop ecx
  pop ebx

  add esp, 8      ;// Освобождаем выделенную память
  pop ebp         ;// Восстанавливаем старый кадр стека
  ;// EAX - кол-во тактов выполнения 2 PUSH + LFENCE
end;
```

Рисунок 20 – Получение кол-во тиков погрешности, затрачиваемое на команды RDTSC и работу со стеком

После генерации матрицы на основе RDTSC, генерируется вторая матрица на основе системного таймера посредством вызова WinAPI QueryPerfomanceCounter(), код генерации, которой изображён на рисунке 21.

```
function FPGenerationPerfCounter(const n, M: Integer): TTimeMatrix;
var
  PrerfFreq: Int64;
  I, J: Integer;
  T1, T2, DiffT: Int64;
  Diff: Double;
begin
  I := 0;
  SetLength(Result, n, M);
  QueryPerformanceFrequency(PrerfFreq);

  while I < n do
  begin
    J := 0;
    // Инициализация параметров ПЕРЕД началом каждой новой строк
    // что бы по одним и тем же параметрам располагались одни и те же значения
    InnerCounter := START_BYTE;
    CurParam := 0;

    // Повторение вызова с разными параметрами указанное кол-во раз
    while J < M do
    begin
      QueryPerformanceCounter(T1);

      // Получение случайного кол-ва байт
      wrapRandBytes();

      QueryPerformanceCounter(T2);
      // Считаем сколько прошло тактов между этими двумя точкам
      DiffT := (T2 - T1);
      // Уник параметры пойдут вправо
      Result[I][J] := DiffT;
      J := J + 1;
    end;
    I := I + 1;
  end;
end;
```

Рисунок 21 – Генерация матрицы выполнения кода на основе WinAPI QueryPerfomanceCounter()

WinAPI функция QueryPerfomanceCounter() возвращает количество тиков системного таймера. Для получения времени, необходимо разделить количество тиков на частоту системного таймера, полученную при вызове WinAPI QueryPerformanceFrequency().

После генерации двух матриц происходит их вычитание между собой по модулю, и результирующая дельта-матрица считается матрицей времени

выполнения кода текущего компьютера. Логика вычитания матриц изображена на рисунке 22.

```
procedure SubtractMatrix(const MatrixA, MatrixB: TTimeMatrix;
  var ResultMatrix: TTimeMatrix);
var
  I, J: Integer;
begin
  if Length(MatrixA) <> Length(MatrixB) then
    raise Exception.Create('Матрицы должны иметь одинаковые размеры');

  SetLength(ResultMatrix, n, M);

  for I := Low(MatrixA) to High(MatrixA) do
  begin
    SetLength(ResultMatrix[I], Length(MatrixA[I]));
    for J := Low(MatrixA[I]) to High(MatrixA[I]) do
    begin
      if Length(MatrixA[I]) <> Length(MatrixB[I]) then
        raise Exception.Create('Размер строк матриц должен совпадать');

      ResultMatrix[I][J] := Abs(MatrixA[I][J] - MatrixB[I][J]);
    end;
  end;
end;

procedure GenAndSaveOnOne1();
begin
  Gen1 := FPGenerationCPU(n, M);
  Gen3 := FPGenerationPerfCounter(n, M);

  // Получаем delta_matrix - GenDif1
  SubtractMatrix(Gen1, Gen3, GenDif1);

  // Сохраняем delta_matrix в файл
  SaveGenToFile(GenDif1, n, M, GEN1_FILE);
end;
```

Рисунок 22 – Генерация двух матриц выполнения кода и нахождение дельта-матрицы

После чего итоговая дельта-матрица записывается в файл. На втором компьютере проводятся точно такие же операции, а затем происходит сравнение между матрицей первого и второго ПК, что изображено на рисунке 23.

```
function GetNumCoincidences(const fp1, fp2: TTimeMatrix;
  const n, M: Integer): Integer;
var
  num_coindences: Integer;
  I, J: Integer;
  fp1_mode: TArray<Double>;
  check: boolean;
  ColumsArr: TArray<Double>;
begin
  num_coindences := 0;
  SetLength(fp1_mode, 50); // Кол-во столбцов
  // Идём по всем столбцам и найдём для каждого mode -
  // наиболее частовстречаемое значение
  for I := 0 to 49 do // проходимся по всем столбцам
  begin
    ColumsArr := GetColumnsAsArray(fp1, I);
    fp1_mode[I] := FindMode(ColumsArr);
  end;

  for I := 0 to 49 do
  begin
    check := FALSE;
    J := 0;
    while (J < 1000) and (not check) do
    begin
      // Проходимся по столбцам
      if fp1_mode[I] = fp2[J][I] then
      begin
        Inc(num_coindences);
        check := True;
      end
      else
        Inc(J);
    end;
  end;
  Result := num_coindences;
end;
```

```
function GetColumnsAsArray(const Matrix: TTimeMatrix;
  ColumnIndex: Integer) : TArray<Double>;
var
  I: Integer;
begin
  SetLength(Result, Length(Matrix));
  for I := Low(Matrix) to High(Matrix) do
    Result[I] := Matrix[I][ColumnIndex];
end;
```

```
function FPCheck(const fp1, fp2: TTimeMatrix;
const n, M: Integer; Trashold: Single): boolean;
var
  NumCon1, NumCon2, SumCon: Integer;
  ResPercentage: Single;
begin
  NumCon1 := GetNumCoincidences(fp1, fp2, n, M);
  NumCon2 := GetNumCoincidences(fp2, fp1, n, M);
  SumCon := NumCon1 + NumCon2;

  WriteLn(Format('FPCheck: NumCon1 = %d, ' + //
    'NumCon2 = %d, SumCon = %d', //
    [NumCon1, NumCon2, SumCon]));

  ResPercentage := SumCon / (50 * 2) * 100;

  WriteLn(Format('FPCheck: ResPercentage = %f' + //
  ' Trashold = %f', [ResPercentage, Trashold]));

  Result := ResPercentage >= Trashold;
end;
```

Рисунок 23 – Нахождение mode в матрицах и сравнение матриц между собой

Стоит отметить, что параметры для измеряемого метода CryptGentRandom() располагаются в одинаковом порядке для обеих матрицы, что изображено на рисунке 24.

Рисунок 24 – Схема формирования матрицы с учётом входных параметров для измеряемой функции CryptGenRandom()

При этом, один и тот же параметр для измеряемой функции не вызывается два раза подряд. Это сделано для повышения точности измерения т.к. процессор во время выполнения кода может закешировать результат первого выполнения измеряемой функции и выдать его значительно быстрее при повторном вызове функции с предыдущим входным параметром. Таким образом, сперва происходит измерение времени выполнения функции CryptGenRandom() с параметром 10, затем с параметром 11 и т.д. до 49 параметра – в это время заполняется первая строка матрицы. Как только достигли 49-го параметрами, происходит переход на вторую строку матрицы и снова измеряется функция с параметром 10 и т.д.

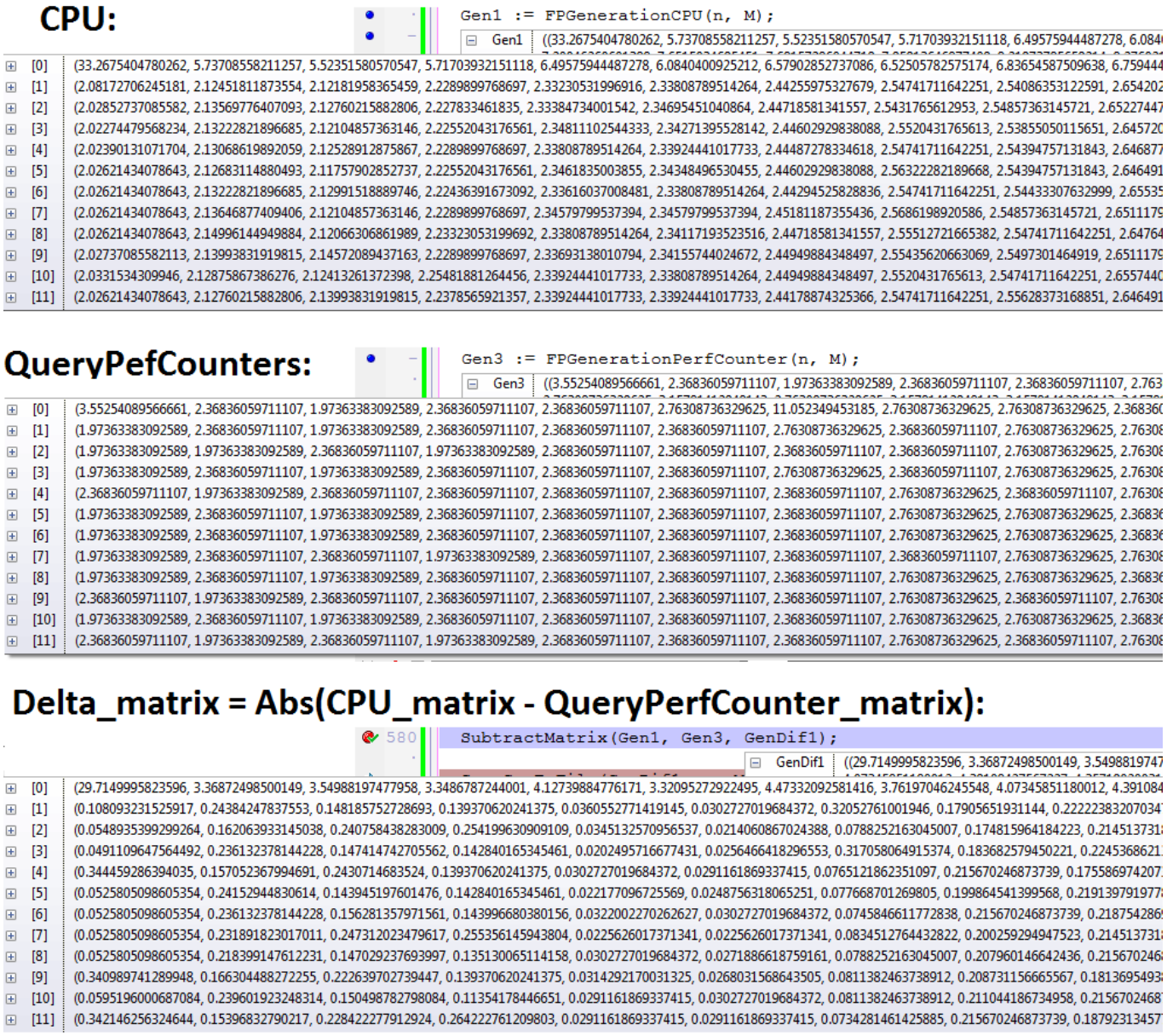


Рисунок 25 – Пример генерации двух матриц и нахождения дельта-матрицы

На рисунке 25 изображены сгенерированные матрицы посредством RDTSC и QueryPerfomanceCounter(), а так же полученная дельта-матрица в результате вычитания матриц по модулю. Значения отображены в микросекундах.

При сравнении матриц происходит расчёт mode- наиболее часто встречаемого значения в столбце с одинаковыми входными параметрами, что изображено на рисунке 26.

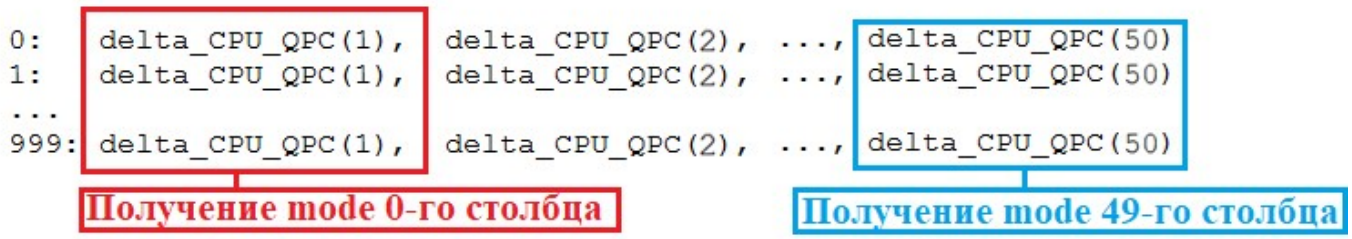


Рисунок 26 – Схема расчёта mode для столбцов дельта-матрицы

После расчёта mode для первой матрицы, происходит поиск значений mode в тех же столбцах второй матрицы. Если значение mode нулевого столбца первой матрицы было обнаружено в нулевом столбце второй матрицы, то увеличиваем счётчик совпадений матриц и переходим к поиску mode следующего столбца. После обработки mode первой матрицы, происходит расчёт mode для столбцов второй матрицы и поиск этих mode в первой матрице. В конце происходит проверка количества совпадений обеих матриц и если процент совпадений больше 50%, то считается что обе матрицы принадлежат к одному и тому же ПК. [7]

В итоге, после многочисленных тестов генерации дельта-матриц, по-прежнему наблюдалось отсутствие стабильности при определении принадлежности матриц к одному и тому же ПК. Было обнаружено, что результаты принадлежности матриц меняются в зависимости от текущей нагрузки на процессор, чего происходить, согласно статье Искандера, не должно.

Стабилизация результатов динамической идентификации

В ОС Windows доступно несколько различных таймеров для использования:

1. RTC с частотой 32.768 КГц.
2. PIT с частотой 1.193180 МГц.
3. ACPI с частотой 3.58 МГц.
4. HPET с частотой от 10 до 24 МГц.
5. LAPIC без фиксированной частоты. [12]

Если процессор поддерживает инвариантный счётчик тиков iTSC в RDTSC, то внутри функции QPC() будет происходить вызов инструкций RDTSC или RDTSCP, что изображено на рисунке 27. Более того, значение iTSC в этом случае будет калиброваться по HPET.

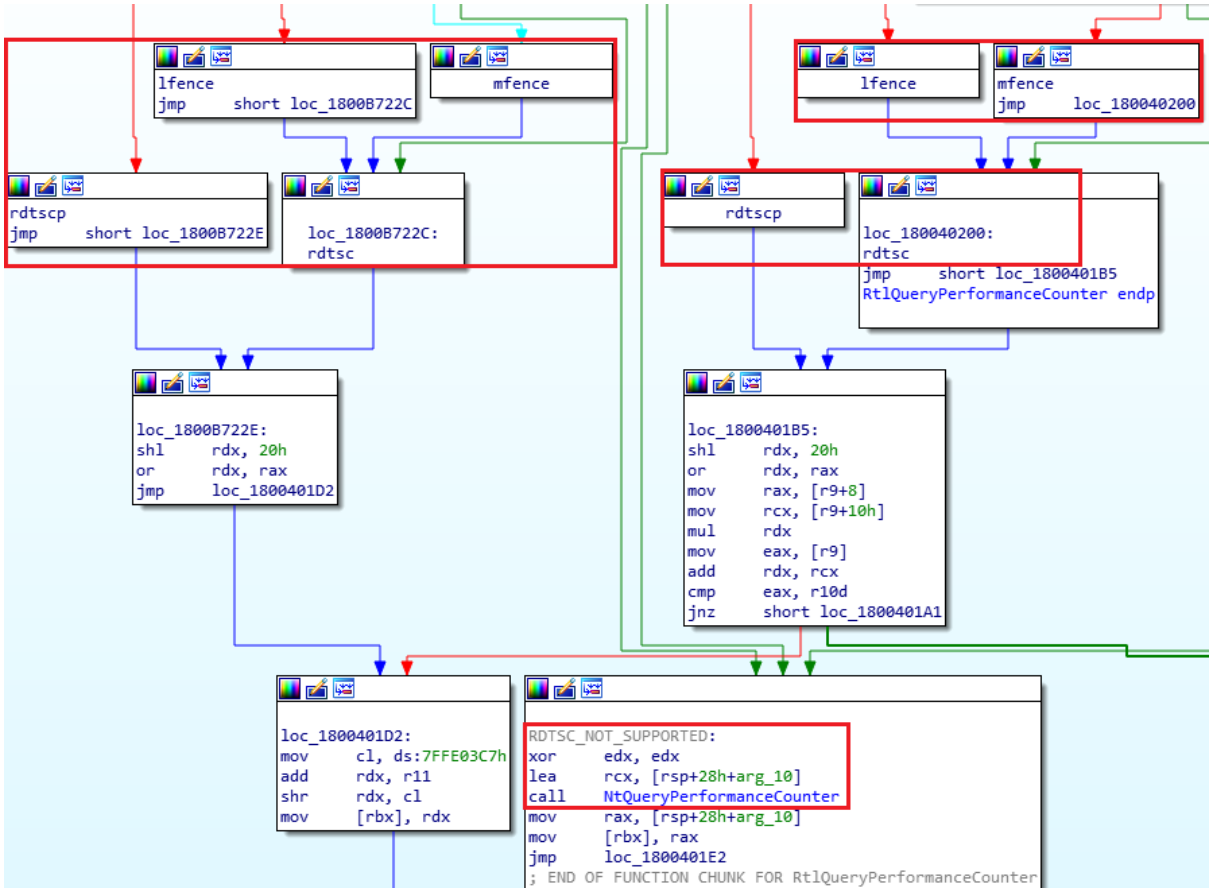


Рисунок 27 – Дизассемблированный листинг функции ntdll.RtlQueryPerformanceCounter

Если iTSC не поддерживается процессором, то QPC() вызовет функцию режима ядра NtQueryPerformanceCounter() которая будет ссылаться на один из доступных аппаратных таймеров – HPET или ACPI.

Согласно полученным рекомендациям от авторов научных статей Искандера и Мигеля, для генерации матриц необходимо использовать разные таймеры с отличающимися частотами. На текущий момент RDTSC и QPC() ссылаются на один и тот же счётчик iTSC. Чтобы переключить QPC на HPET, необходимо выполнить следующую команду в cmd.exe от имени администратора:

bcdedit /set useplatformclock true

и затем перезагрузить ПК [17]. Следует так же убедиться, что в UEFI/BIOS разрешено использование таймера HPET, настройка которого изображена на рисунке 28.

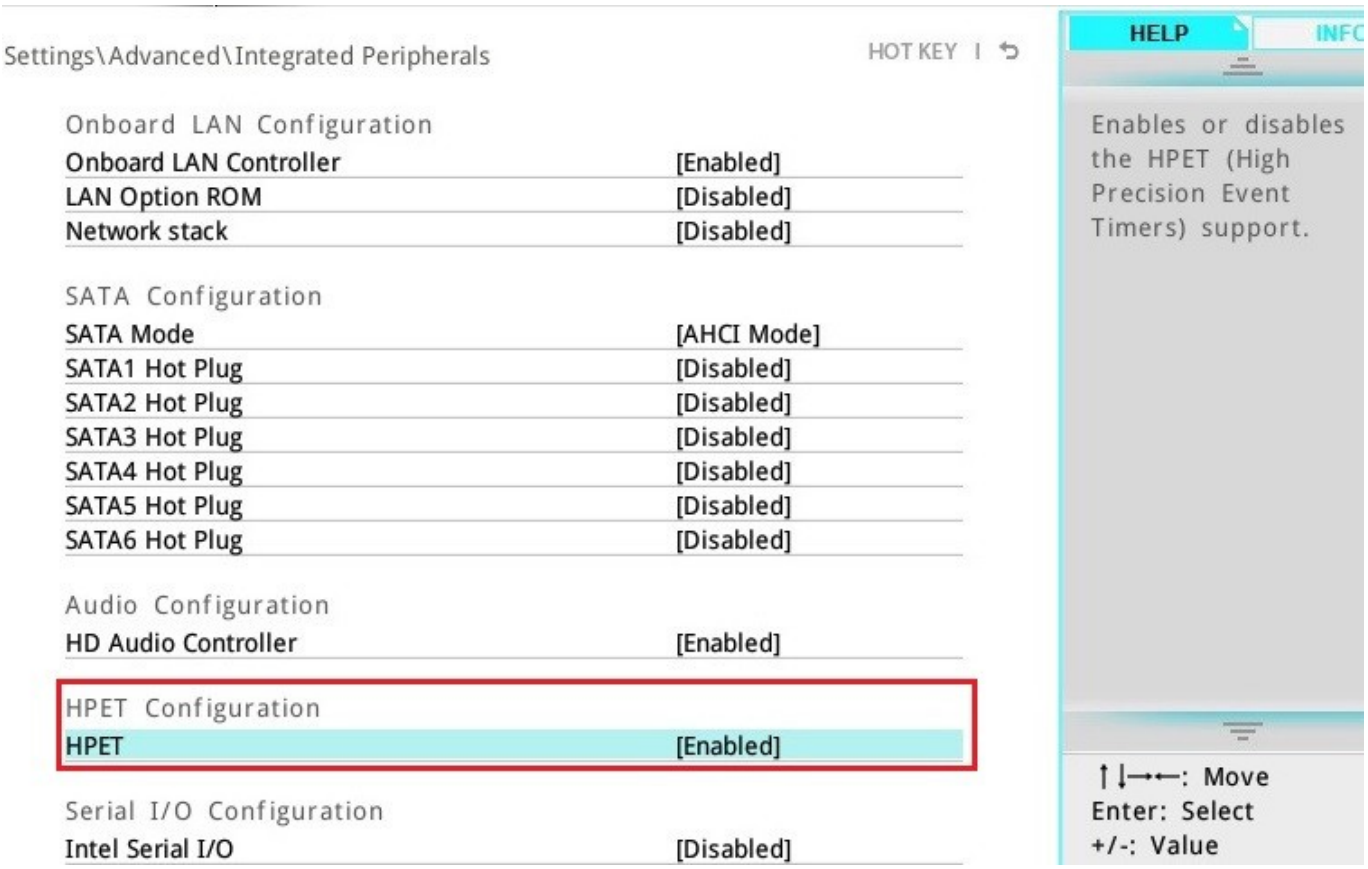


Рисунок 28 – Настройка активации HPET в UEFI/BIOS

Если выполнить команду **bcdedit useplatformclock** с отключенным HPET в UEFI/BIOS, то после перезагрузки ОС QPC() будет использовать ACPI таймер. Для возврата на использование iTSC в QPC(), необходимо выполнить следующую команду в cmd.exe от имени администратора:

bcdedit /set useplatformclock false

и затем перезагрузить ПК.

Чтобы проверить, изменился ли таймер, используемый в QPC(), необходимо запросить частоту системного таймера через WinAPI QueryPerfomanceFrequency() или посмотреть частоты QPC в утилите «CPU-Z» [18], что демонстрируется на рисунке 29.

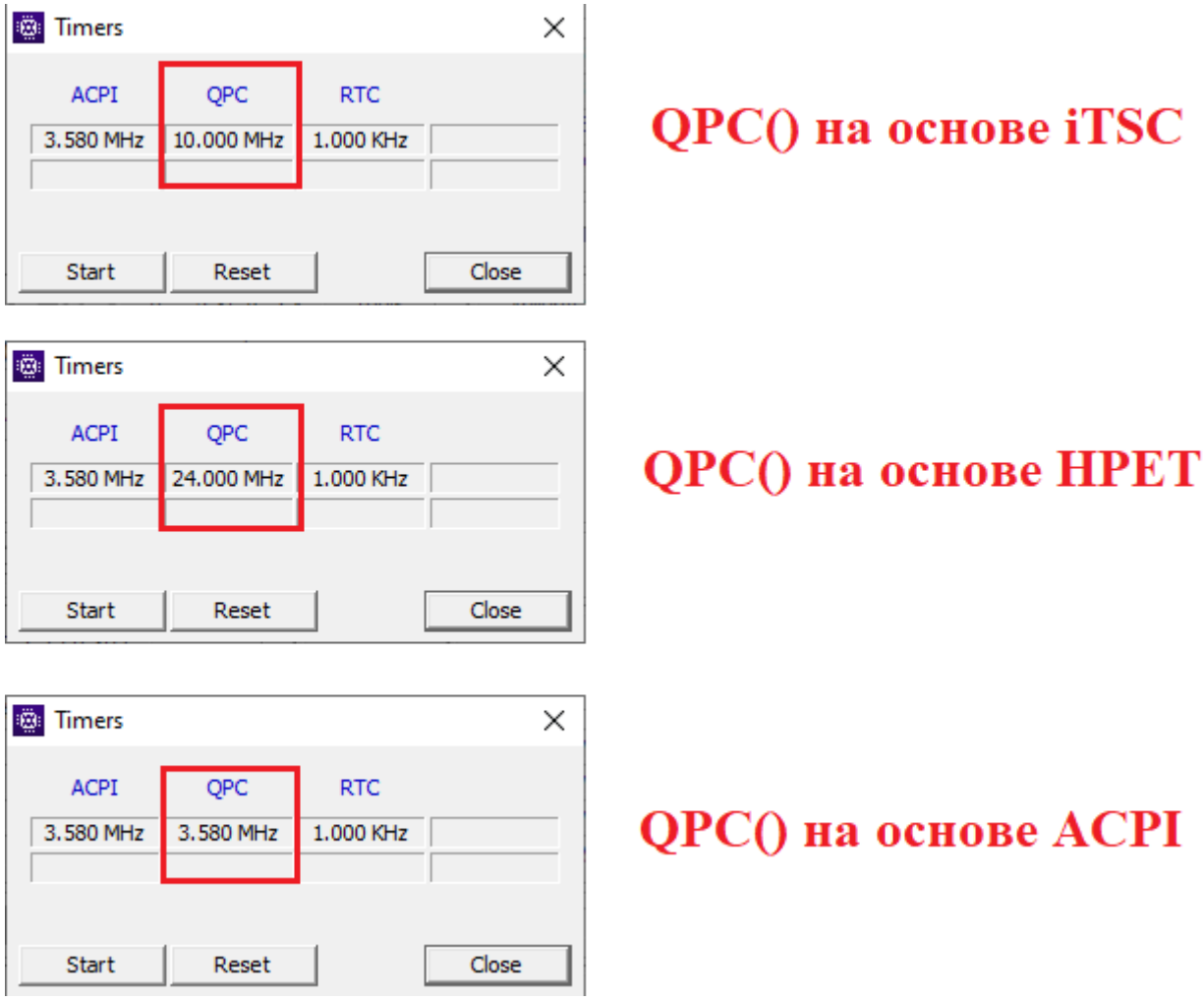


Рисунок 29 – Различия частот таймеров для QPC в программе «CPU-Z»

При использовании HPET или ACPI в качестве основы QPC было замечено ухудшение отзывчивости работы ОС. Это связано с временными задержками, необходимыми для получения значения таймеров HPET или ACPI, когда как получение значения iTSC выполняется почти мгновенно.

В ОС Windows в режиме пользователя доступно ограниченное количество функций для измерения времени выполнения, ссылающиеся на таймеры с разной частотой. Основные источники для измерения времени выполнения кода:

1. GetSystemTimeAdjustment() – возвращает интервал программного таймера ОС который равен 15.625 мсек с частотой 1 / 0.015625 = 64 Гц. Производные функции с идентичной частотой:
   - GetTickCount()
   - GetSystemTime()
   - timeGetTime()

   Данные функции не могут быть использованы для измерения скорости выполнения участок кода, т.к. работают на слишком малой частоте.

2. QueryPerformanceFrequency() – использует iTSC с виртуальной частотой, рассчитываемой динамически исходя из аппаратных возможностей (в данном случае 10 МГц), а так же таймер HPET (от 10 до 24 МГц) или ACPI (3.59 МГц).
3. Ассемблерная инструкция RDTSC или RDTSCP для получения iTSC.

Для расчета двух матриц использовался QPC() на основе таймера HPET и RDTSC на основе iTSC. Тестовое консольное приложение будет скомпилировано под x32 для применения ассемблерных вставок с RDTSC.

Локальное тестирование на одном ПК происходило путём генерации двух матриц, нахождения дельта матрицы и сохранения её в файл Gen1.txt рядом с программой. После чего происходило 5-ти секундное ожидание и последующая генерация двух матриц выполнения кода, рассчитывалась дельта-матрица и сохранялась в файл Ge2.txt. Затем происходила загрузка Gen1.txt и Gen2.txt файлов и

сравнение матриц для определения, принадлежат ли обе матрицы одному и тому же ПК. Данная процедура повторялась 10 раз подряд. В итоге, локальные тесты на одном ПК показывали 100% совпадение дельта-матриц в 10 случаев из 10 при отсутствии какой-либо нагрузки на процессор. Однако, если сгенерировать дельта-матрицу без нагрузки на процессор, а затем сгенерировать дельта-матрицу при одновременно запущенном стресс тесте процессора и сравнивать обе дельта-матрицы, то результат проверки идентичности матриц может варьироваться от 40% до 90% совпадений, что является нестабильным результатом. Для стресс тестов нагрузки процессора использовалась утилита «CPU Stress» и функция «Stress CPU» в программе «CPU-Z».

В своей статье Искандер описывал стабильную генерацию матриц даже при 100% нагрузке на процессор, чего не удалось повторить в текущей работе.

Для повышения стабильности генерации матриц были предприняты следующие попытки:

1. Замена RDTSC на более современную ассемблерную инструкцию RDTSCP. Тестировалась так же замена команды LFENCE на CPUID, MFENCE для ожидания исполнения конвейера/очистки конвейера команд CPU.
2. Компиляция под x64 для прямых вызовов WinAPI минуя прослойку WoW64. При переходе на x64 потребовалось отказаться от использования встраиваемых ассемблерных вставок, т.к. в x64 Delphi они запрещены. Вместо вставок используется обычный вызов ассемблерных функций замера времени.
3. Тестирование различных вариаций входных параметров (генерируемых байт) функции CryptGenRandom(), а так же изменение шага параметров между вызовами.
4. Смена измеряемой функции с CryptGenRadom() на FNV1Hash() и BobJenkinsHash() с заранее сгенерированными входными значениями.
5. Повышение точности работы QPC() посредством вызова WinAPI timeBeginPeriod(1)
6. Вызов WinAPI Sleep(1) перед каждым замером времени выполнения кода для уменьшения влияния нагрузки CPU на результат. Использование нулевого

аргумента не влияло на результат, а использование единицы в качестве аргумента значительно удлиняло время генерации матриц и незначительно стабилизировало результат.

7. Смена подхода к генерации матриц, при котором заполнялись обе матрицы сразу, а не по отдельности (после заполнения нулевой строки нулевого столбца матрицы RDTSC/P происходило измерение QPC() и заполнение нулевой строки нулевого столбца матрицы QPC()).
8. Отключение в UEFI ПК с Windows 10 следующих технологий процессора:
    - Hyper-threading
    - TurboBoost
    - DVFS
    - Enhanced Intel SpeedStep

Однако достичь стабилизации при сравнении локальных дельта-матриц не удалось. По-прежнему, даже после выполнения всех вышеописанных пунктов, матрица, сгенерированная без нагрузки на CPU и с нагрузкой, считается двумя мало похожими матрицами (процент идентичности двух локальных матриц по-прежнему варьируется около 70%), что демонстрируется на рисунке 30.

Рисунок 30 – Сравнение полученной и ожидаемой стабильности дельта-матриц

Стоит отметить, что тестирование генераций и сравнение матриц выполнялось лишь на двух ПК с различными аппаратными и программными средствами. Более того, ПК с ОС Windows 7 являлся ноутбуком с заблокированным производителем BIOS (отсутствовала возможность настраивать процессор, ОЗУ и т.д.).

Рекомендации по дальнейшему исследованию

Несмотря на то, что в данной работе не удалось реализовать стабильную генерацию матриц скорости выполнения кода для динамической идентификации ПК, рекомендуется продолжать попытки реализации описанного механизма с учётом опыта текущей работы, а именно:

1. Необходимо выполнять тестирование генерации на нескольких ПК с одинаковыми аппаратными и программными компонентами. В данной работе этого не было сделано, т.к. не удалось получить разрешение от университета на эксперименты с учебными ПК.
2. Работать с таймерами HPET и ACPI напрямую из режима ядра, не ограничиваясь режимом пользователя. В данной работе использовались лишь WinAPI функции режима пользователя.
3. Протестировать генерацию матриц на основе HPET и ACPI из ядра ОС без использования RDTSC/RDTSCP.
4. Стабилизировать частоту работы оперативной памяти путём настроек в BIOS/UEFI.
5. Попробовать применить STM для исключения прерывания планировщика ОС при работе с памятью.
6. Проводить тестирования на различных процессорах Intel AMD. Существует вероятность, что данный подход может стабильно работать только на определённых моделях/сериях процессоров. Однако Искандер не смог вспомнить, на ПК с какими процессорами выполнялось его тестирование.

Список используемых источников

1. Sanchez-Rola, I. Santos, D. Balzarotti, Clock Around the Clock: Time-Based Device Fingerprinting [Электронный ресурс] URL: https://www.s3.eurecom.fr/docs/ccs18_iskander.pdf (дата обращения: 15.04.2025)

2. Pedro Miguel S., José Maria J, Alberto Huertas C, A methodology to identify identical single-board computers based on hardware behavior fingerprinting [Электронный ресурс] URL: https://www.sciencedirect.com/science/article/pii/S108480452200220X (дата обращения: 16.04.2025)

3. Btbd, HWID spoofer for Windows [Электронный ресурс] URL: https://github.com/btbd/hwid (дата обращения: 20.04.2025)

4. SamuelTulach, Manual map kernel mode driver [Электронный ресурс] URL: https://github.com/SamuelTulach/nullmap (дата обращения: 20.04.2025)

5. Samuel Tulach, Windows kernel-mode hardware identifier (HWID) spoofer [Электронный ресурс] URL: https://github.com/SamuelTulach/mutante (дата обращения: 20.04.2025)

6. Marylin, Системные таймеры. Часть[6] - Основы профилирования кода [Электронный ресурс] URL: https://codeby.net/threads/sistemnyye-taimery-chast-6-osnovy-profilirovaniya-koda.74153/ (дата обращения: 20.04.2025)

7. apexlegends, All methods of retrieving unique identifiers(HWIDs) on your PC [Электронный ресурс] URL: https://www.unknowncheats.me/forum/anti-cheat-bypass/333662-methods-retrieving-unique-identifiers-hwids-pc.html (дата обращения: 20.04.2025)

8. Intel Corporation, Using the RDTSC Instruction for Performance Monitoring [Электронный ресурс] URL: https://www.cs.usfca.edu/~cruse/cs210s06/rdtscpm1-1.pdf (дата обращения: 16.04.2025)

9. MSDN BCDEdit /set [Электронный ресурс] URL: https://learn.microsoft.com/ru-ru/windows-hardware/drivers/devtest/bcdedit--set (дата обращения: 01.05.2025).
10. CPU-Z [Электронный ресурс] URL: https://www.cpuid.com/softwares/cpu-z.html (дата обращения: 01.05.2025).
11. RW Utility [Электронный ресурс] URL: https://rweverything.com (дата обращения: 01.05.2025).