This paper contains attempts at practical implementation of PC hardware identification based on code execution speed matrices based on the work" Clock Around the Clock: Time-Based Device Fingerprinting " by Iskander Sanchez-Rola

Table of contents

Justification and implementation of dynamic identification of PC hardware based on the code execution speed matrix

Dynamic identification of PC hardware is an alternative method of identifying PCs based on the measurement of some unique parameters, the combination of which can uniquely identify many PCs with the same hardware and software. Such HWID generation is more difficult to detect and circumvent, because there is no explicit reference to any hardware attribute.

In the 2018 paper" Clock Around the Clock: Time-Based Device Fingerprinting", a group of researchers led by Iskander Sancherz-Rol demonstrated and described a method based on multiple measurement of code execution speed [7]. The article claims that the generated measurements (about 50 thousand calls to the same function) will be unique for each PC due to differences in quartz crystals due to the peculiarities of production. The PC's internal clock uses oscillators based on quartz crystals, and small fluctuations in these crystals can lead to extremely small but measurable differences in clock frequency. Many factors can affect the measurement of code execution speed:

1. Cache misses/TLB and sharing pipeline resources with other CPU threads.
2. Using Hyper-threading
3. Applying Dynamic CPU Frequency and Voltage Scaling (DVFS)

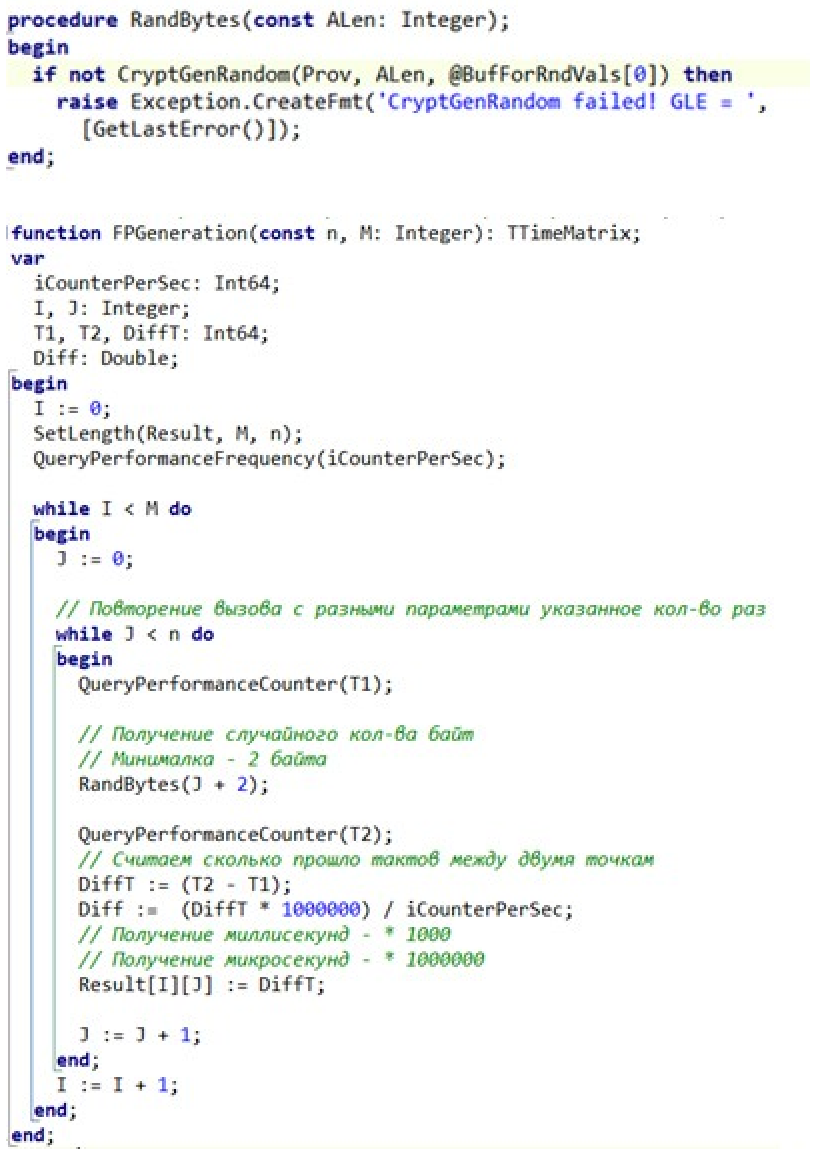
Due to these uncertain factors, one measurement is not enough to get a stable result of the execution speed.

After generating a speed matrix for executing the same function with different input parameters, it is compared with another matrix and determined whether these matrices belong to the same PC or not. The authors implemented a native version for execution on a PC (C programming language) and a version for execution in a browser (JavaScript programming language).Both versions are called CryptoFP because they measure the speed of execution of the CryptGenRandom () function.

In addition to dynamically measuring code execution time, a distinctive feature of this approach is that there is no strict comparison of two identifiers for similarity. Instead, the identity threshold of the two matrices is calculated, and if it exceeds 50%, then both matrices belong to the same PC.

First attempt to implement dynamic identification

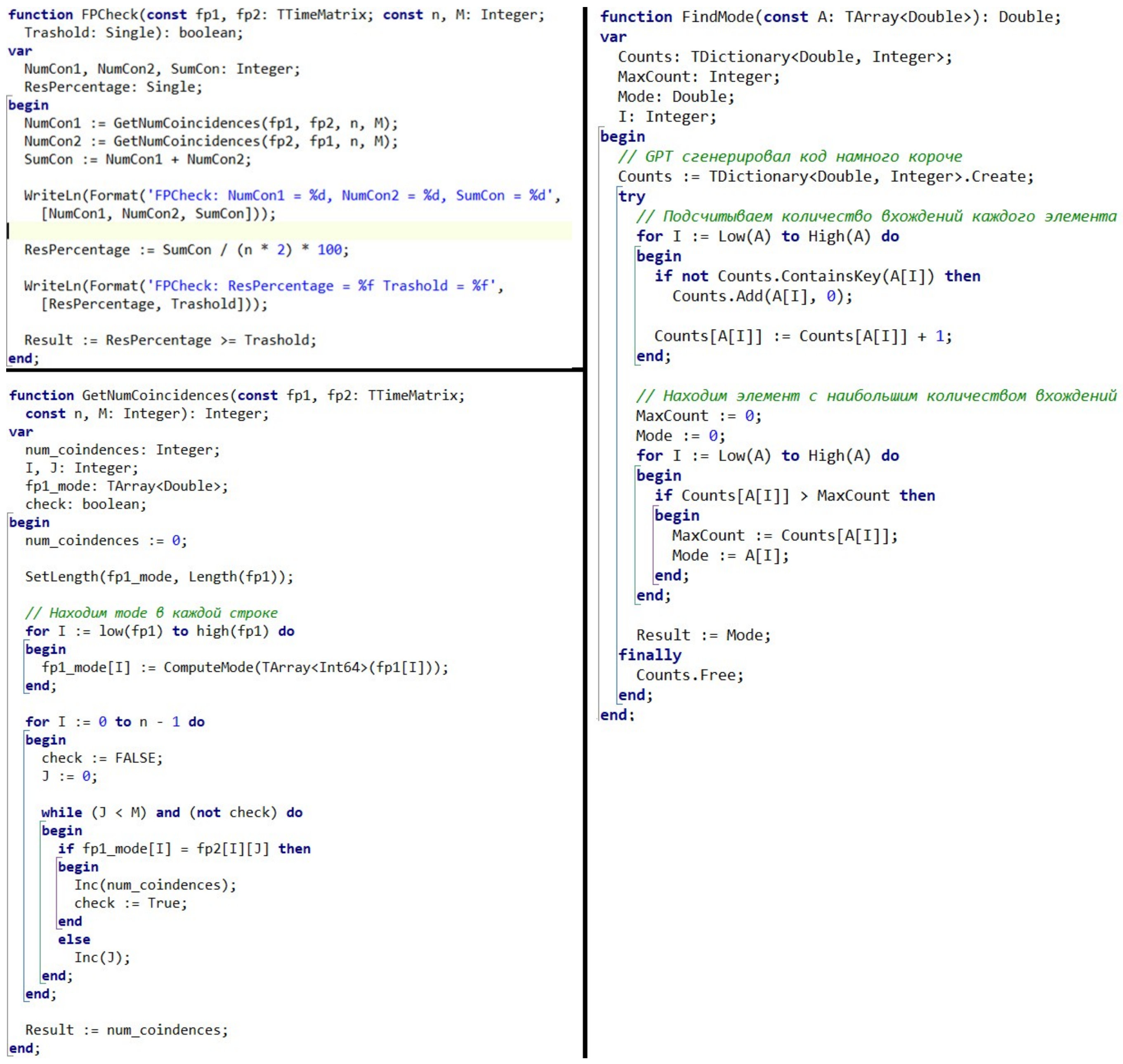
A prototype was implemented to test the described mechanism in the Delphi programming language. Figure 11 shows the code for generating the speed matrix for the RandBytes () function. Inside the RandBytes() function, CryptGenRandom () is called, filling the buffer with cryptographically random bytes.



```
procedure RandBytes(const ALen: Integer);
begin
  if not CryptGenRandom(Prov, ALen, @BuffForRndVals[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;
```

Figure 11-Algorithm for generating a runtime matrix

After the generation stage, methods were implemented for obtaining the mode of each row of the matrix (finding the most repeated values in the matrix row to get rid of interference and switching the OS scheduler), which is shown in Figure 12. Figure 12 also shows an algorithm for determining the identity of matrices based on calculating the sum of occurrences of mode rows of two matrices and applying a threshold of 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;
```

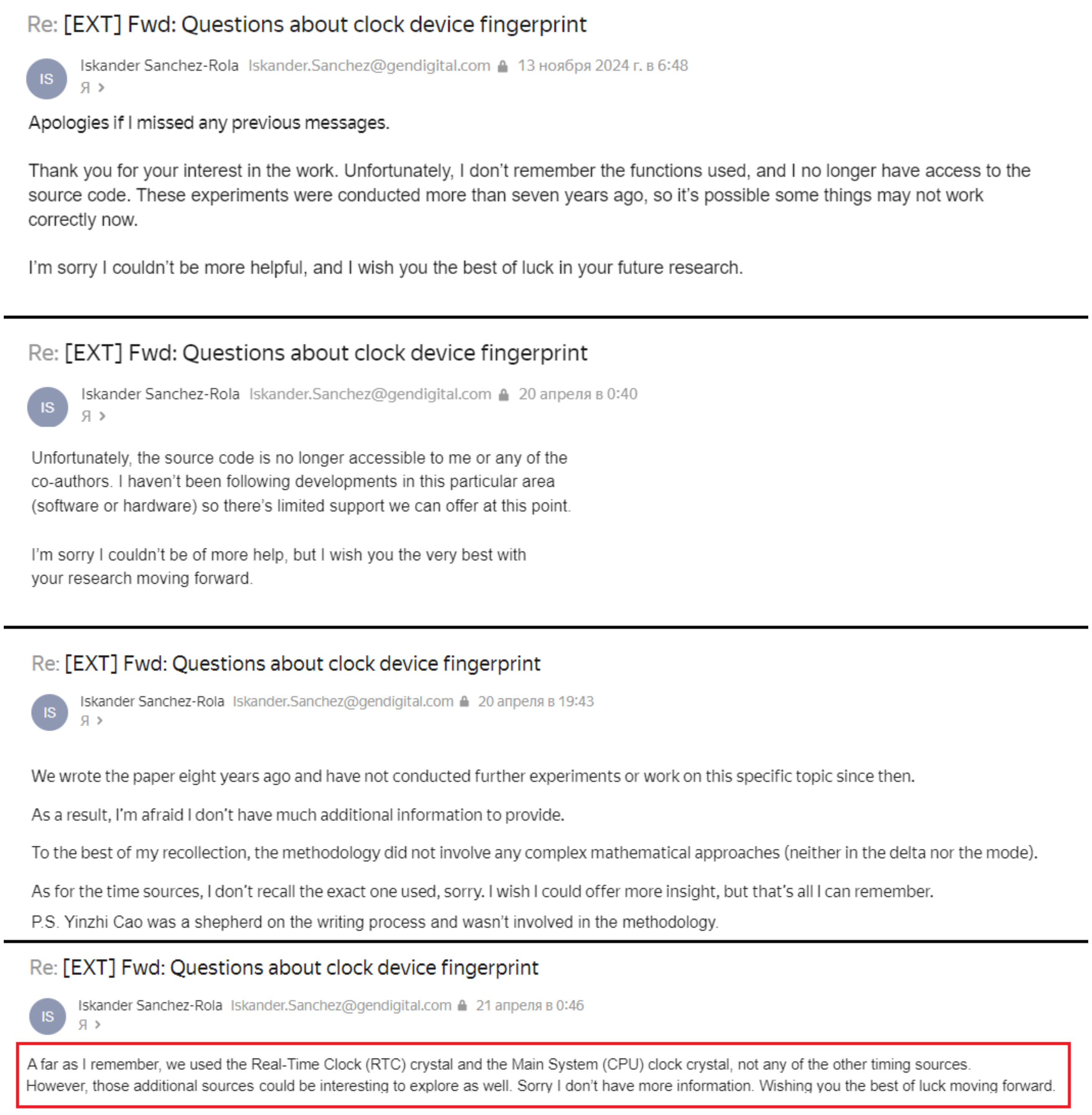
```
function FindMode(const A: TArray<Double>): Double;
var
  Counts: TDictionary<Double, Integer>;
  MaxCount: Integer;
  Mode: Double;
  I: Integer;
begin
  // GPT сгенерировал код намного короче
  Counts := TDictionary<Double, Integer>.Create;
  try
    // Подсчитываем количество вхождений каждого 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;
```

Figure 12-Algorithm for finding the calculated mode of the matrix fp1 in the matrix fp2 and the function for finding the mode of the matrix row

During the development of the prototype, several time measurement options were tested, but it was not possible to achieve unambiguous performance. It was suggested that Iskander's article lacks some of the key information needed to implement a working version of CryptoFP.

Clarifying key details

After unsuccessful attempts to reproduce the stable result of generating the execution matrix, letters were sent to the authors of this article to clarify key details. Figure 13 shows Iskander's answers to questions regarding the implementation of the native version of code execution speed matrix generation – CryptoFP.



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

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

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

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

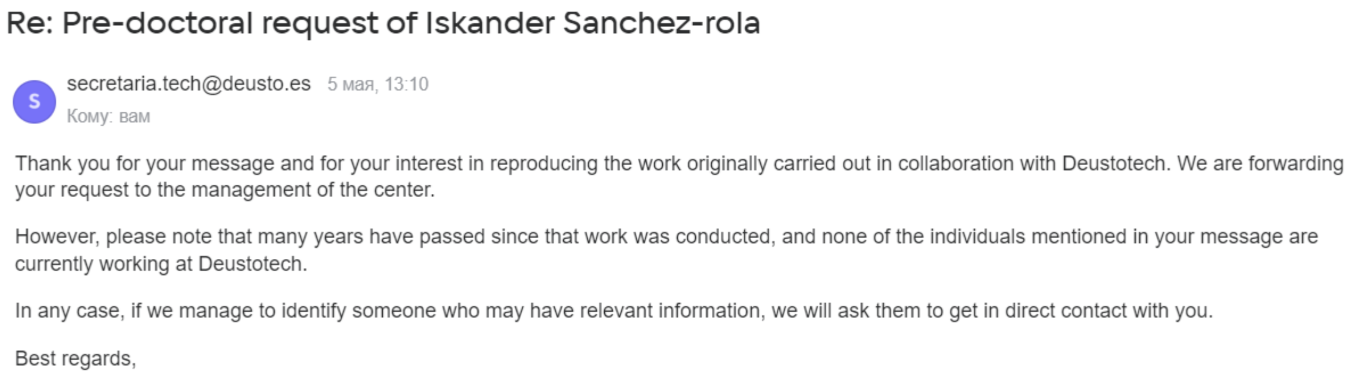
Iskander Sanchez-Rola Iskander.Sanchez@gendigital.com 21 апреля в 0:46
Я >

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.

Figure 13-Iskander Sanchez-Rola's email Responses to CryptoFP Implementation Questions

From Iskander's answers, it became clear that access to the source code of the native version of the CryptoFP application was lost, and key details were forgotten. However, it was confirmed that the standard Windows API for working with date and time was used to measure the code execution time, and work with GPU timers was also excluded.

A request was also sent to the University of Deusto in the Bilbao campus, where the author of the article in 2018 conducted experiments on a large number of PCs with the same hardware components. Figure 14 shows a response from the Bilbao campus with a promise to help with obtaining information about the characteristics of PCs on which the native version of CryptoFP was tested in 2018, but no further communication followed.


Re: Pre-doctoral request of Iskander Sanchez-rola

secretaria.tech@deusto.es 5 мая, 13:10
Кому: вам

Thank you for your message and for your interest in reproducing the work originally carried out in collaboration with Deustotech. We are forwarding your request to the management of the center.

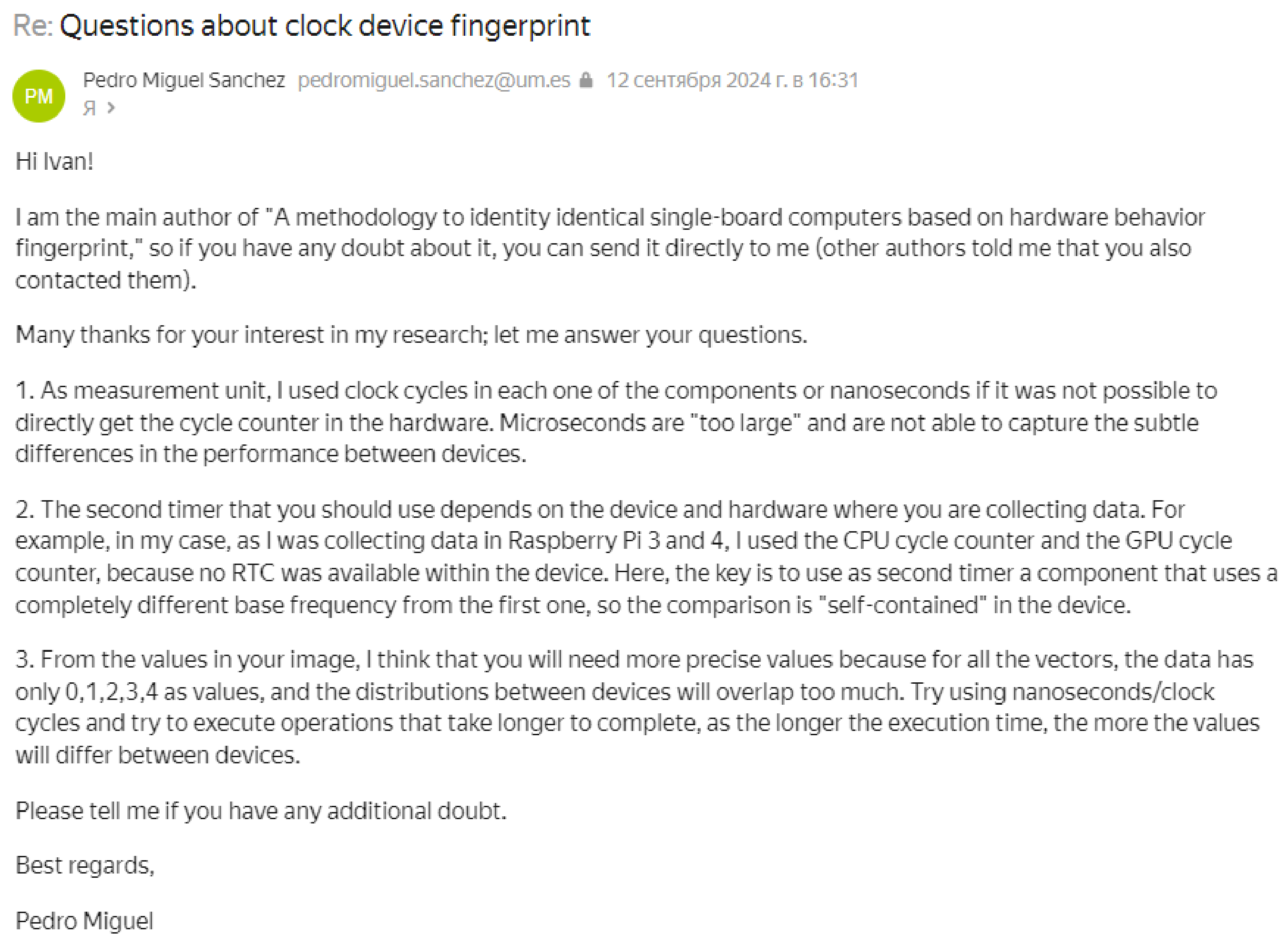
However, please note that many years have passed since that work was conducted, and none of the individuals mentioned in your message are currently working at Deustotech.

In any case, if we manage to identify someone who may have relevant information, we will ask them to get in direct contact with you.

Best regards,

Figure 14-Response of the University of Deusto Bilbao Campus by email to a request for information about the characteristics of the PC and the CryptoFP source code

In the course of analyzing other scientific papers referring to Iskander's article, we managed to contact the author of the 2023 study "A methodology to identity identical single-board computers based on hardware behavior fingerprint" [8], Pedro Miguel Sanchez, who also tried to reproduce the device identification mechanism based on generating a code execution speed matrix. Figure 15 shows Pedro Miguel Sanchez's answer to the CryptoFP implementation questions.


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

Figure 15-Pedro Miguel Sanchez's email response to the implementation of CryptoFP Code Execution Matrix Generation

Pedro Miguel Sanchez said that for the mechanism to work properly, it is necessary to compare the code execution results of two different devices with quartz crystals, and their accuracy must be extremely high (it is necessary to compare the number of code execution cycles, not milliseconds and microseconds).

In an article by Pedro Miguel Sanchez, the Iskander mechanism was shown to be inoperable due to the use of specific hardware (Raspberry Pi 4 Model B and Raspberry Pi 3 Model B), which does not have a second accurate timer (RTC) required for generating and comparing matrices [8].

After putting the authors ' answers together, the following conclusions were drawn:

1. To measure the speed of code execution, we used the standard Windows API – Iskander.

2. Interaction with GPU timers via CUDA / OpenCL did not occur-Iskander.
3. You must use at least two different timers with different frequencies. One timer is not enough for a stable result. The more different the frequency of the timers, the better – Miguel.
4. The more accurate both timers are, the better. It is recommended to work with nanoseconds or processor ticks to measure the speed of code execution-Miguel.
5. RTC and processor timer/system timer – Iskander were used. The author does not fully remember the details and is confused in the concepts of timers.
6. There is no key data on the characteristics of the PC on which Iskander tested the stability of the native version of CryptoFP in 2018. The article mentions that testing took place on 100-200 identical Windows 7 and Linux PCs, but there is no information about the processors, the amount of RAM, motherboards and video cards of these PCs in the article.
7. For each timer, you need to generate your own matrix, and then subtract one matrix from the other modulo. The difference of the two matrices will be the device-Miguel matrix.

Second attempt to implement dynamic identification

Taking into account the responses of the authors of scientific articles, it was decided to study and test the method for measuring the speed of code execution based on processor ticks as the most accurate measurement option.

All further tests took place on two PCs, the characteristics of which are presented in Table 7. The development language of the test console application is Delphi 12. Compilation for tests was performed first under x32, and then under x64.

Table 7-PC characteristics for testing

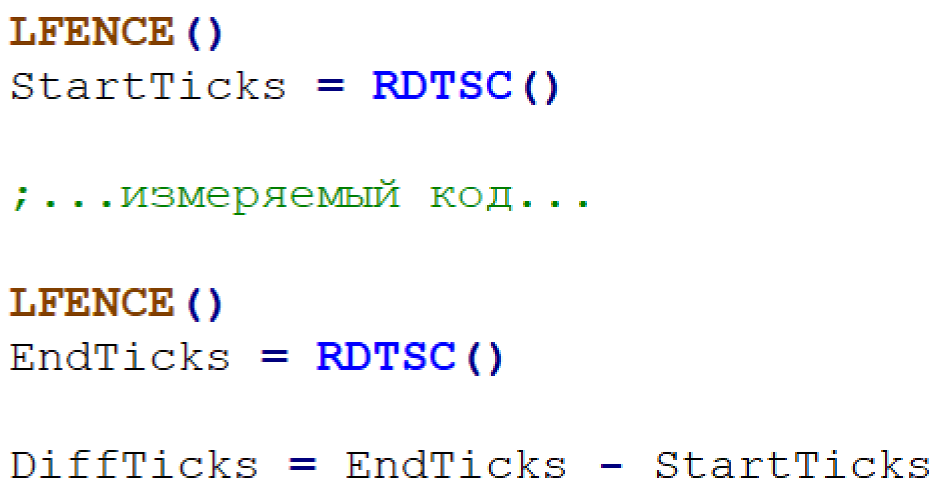
Processor	**Video card**	**RAM**	**Motherboard**	**Disk type where the**	**was located OS**

				program	
Intel Core i5-3230M 2.6 GHz	NVIDIA GT 640M 2 GB	DDR3 6 GB	Acer VA70_HC	SSD	Windows 7 7601 SP1
Intel Core i7-7700 3.6 GHz	NVIDIA GTX 1060 6 GB	DDR4 16 GB	MSI B150 GAMING M3 (MS-7978)	SSD	Windows 10 22H2 19045.5737

To stabilize the results of code execution speed testing, the following steps were performed when running the program under test:

1. Вызывался метод The WinAPI SetProcessAffinityMask() method was called to limit the code execution of all application threads on a single processor core. This is necessary so that during a long measurement of the execution code, the processor does not transfer the task execution to another core, which may affect the final results.
2. Вызывался метод The WinAPI SetPriorityClass() method was called to set the process priority to HIGH_PRIORITY_CLASS and prevent application thread code from being preempted by the OS thread scheduler.

After that, the code execution speed was measured by executing the RDTSC assembler instruction. The RDTSC instruction returns the number of ticks in the EDX and EAX cregisters since the processor was last reset. In modern systems, the RDTSC counter is invariant to iTSC, i.e. it does not depend on the use of energy-saving technologies and increases with a constant frequency. The pseudo-code for using RDTSC is shown in Figure 16.



```
LFENCE()
StartTicks = RDTSC()

;...измеряемый код...

LFENCE()
EndTicks = RDTSC()

DiffTicks = EndTicks - StartTicks
```

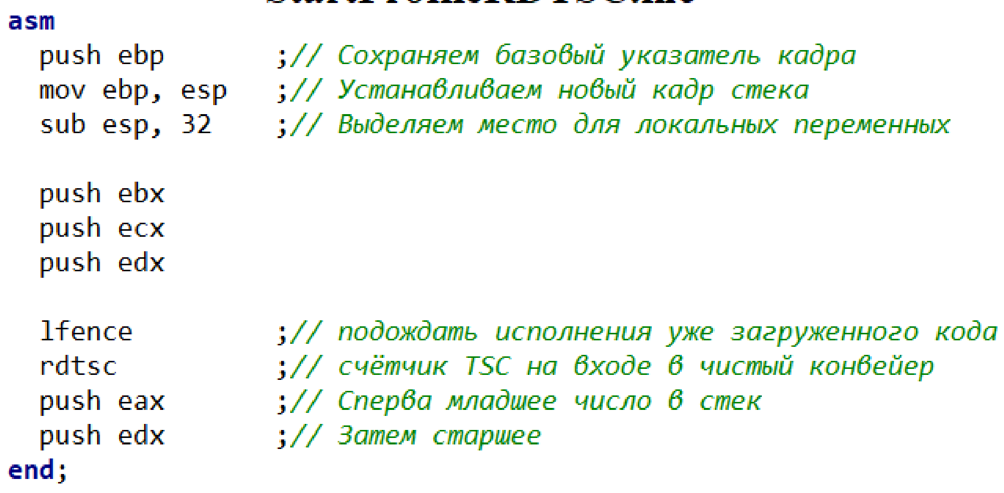
Figure 16-Pseudo-code for using the RDTSC assembly instruction

It should be taken into account that in order to use RDTSC correctly, the processor's instruction pipeline must be reset, otherwise the processor may reorder the execution of commands and RDTSC will be executed after the measured code is called [14]. A processor pipeline is a sequence of steps that each instruction passes through until it is fully executed (decoding, executing, storing the result, etc.).

In this case, the LFENCE assembly instruction is used, but CPUID is also allowedCPUID. The CPUID command forcibly clears the pipeline, and LFENCE waits for all commands to be executed before continuing execution [14].

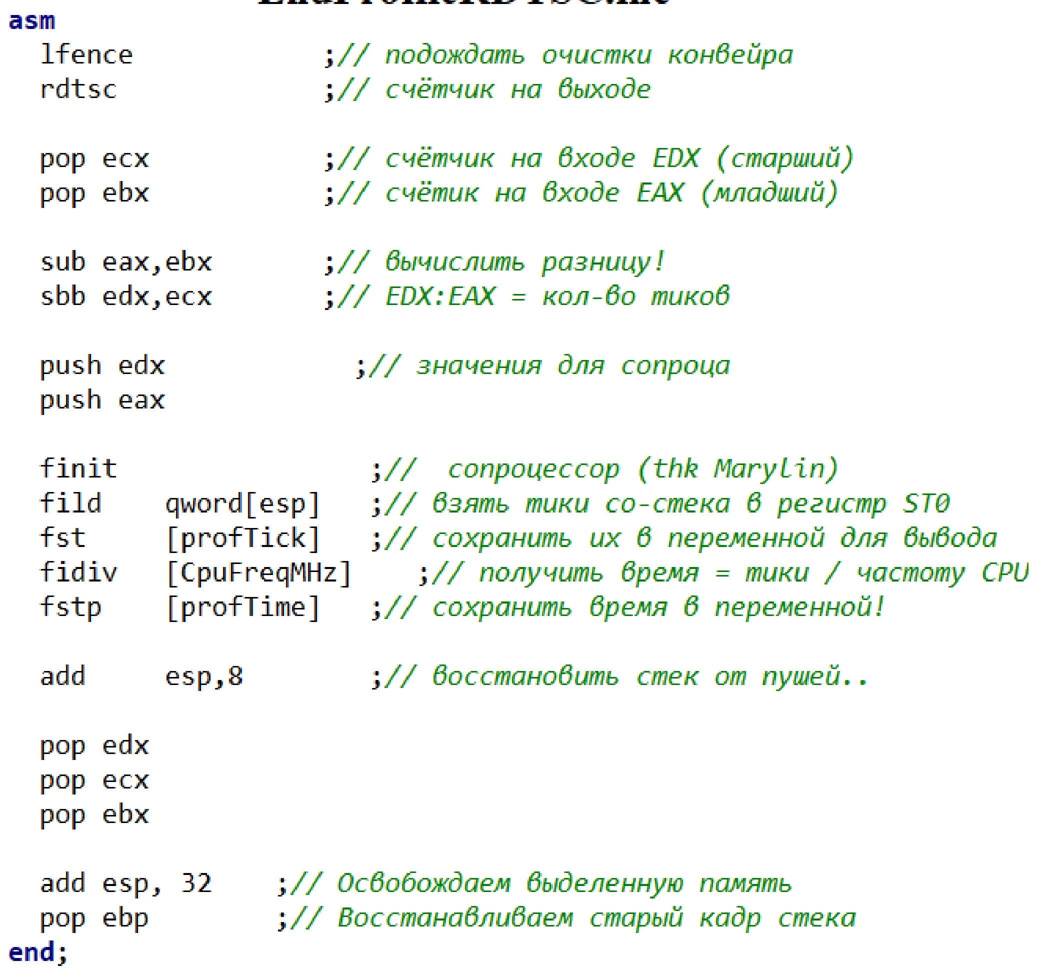
Figure 17 shows a complete assembler list of commands for measuring the speed of code execution time in ticks using the RDTSC command. In StartProfileRDTSC.inc, you work with the stack and get the current value of the processor's tick counter stored on the stack. After that, the measured function CryptGenRandom() is executed. Inc-files are convenient for storing assembler inserts that can be used anywhere in a program compiled under 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;
```

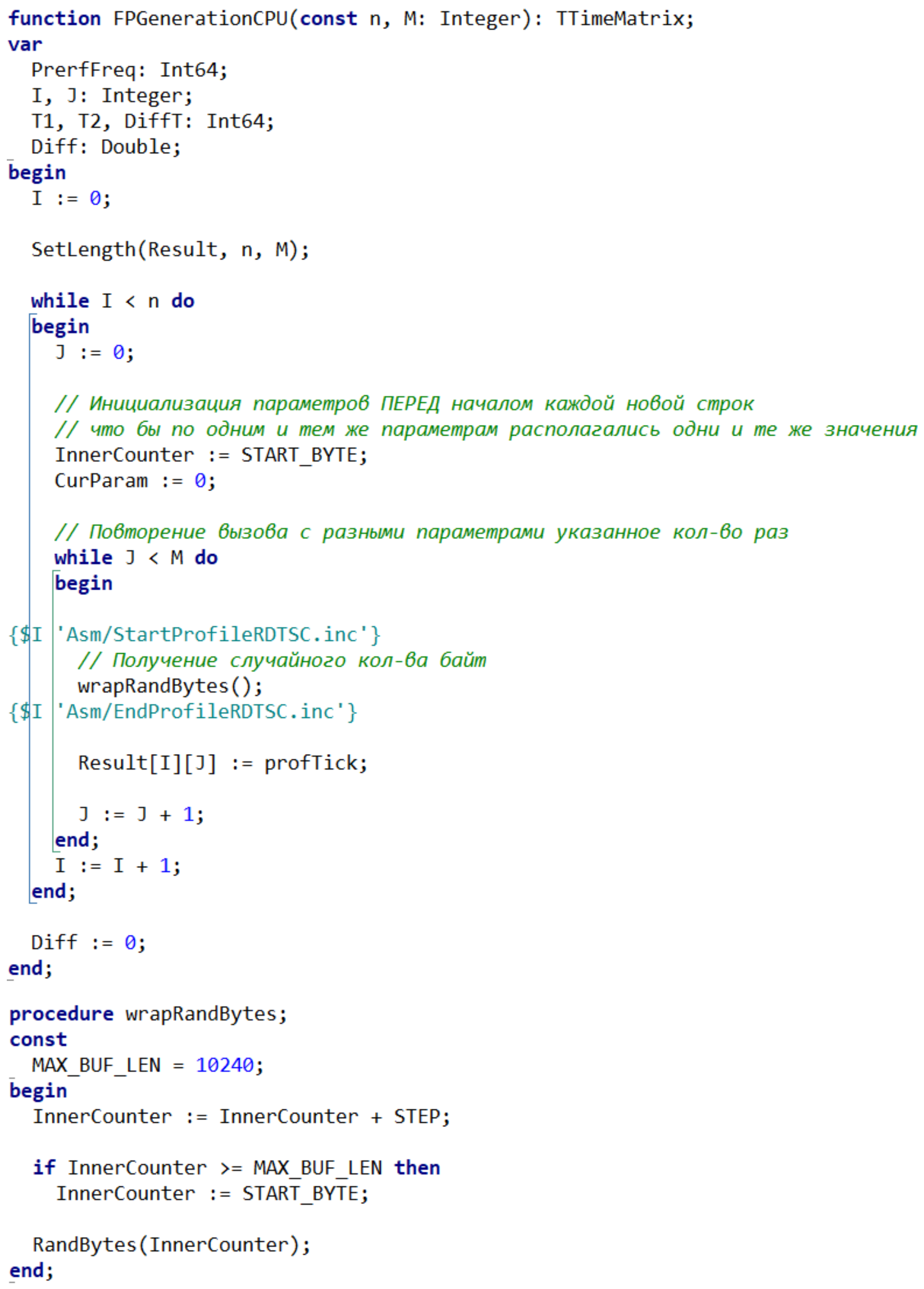
EndProfileRDTSC.inc



```
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;
```

Figure 17-Assembler command listing for measuring code execution speed

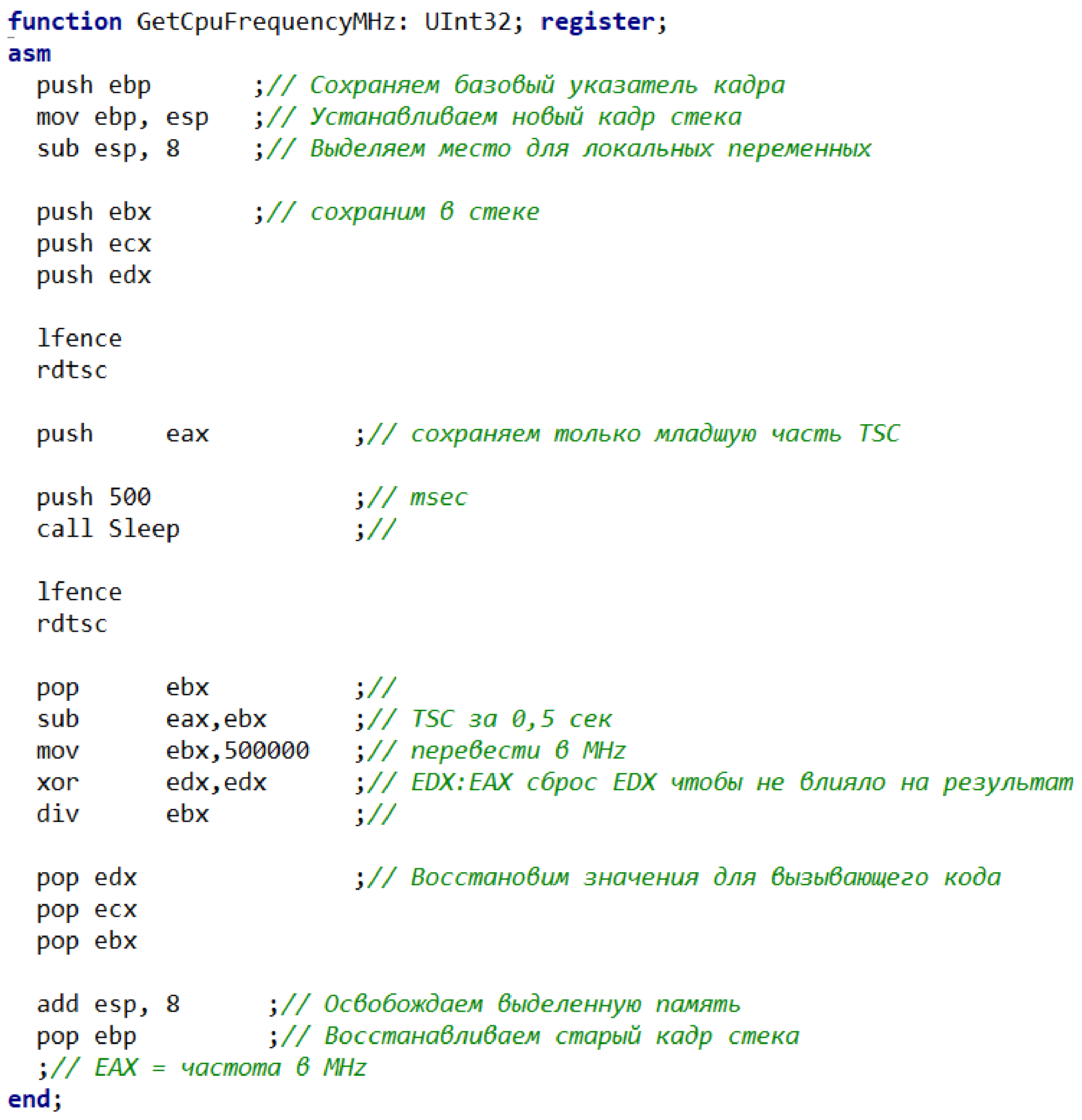
After executing the CryptGenRandom() function,EndProfileRDTSC.inc is called to get the current value of the tick counter, calculate the difference between ticks, and get the execution time by dividing the ticks by the CPU frequency in the coprocessor. Figure 18 shows the generation of a code execution rate matrix with RDTSC ticks preserved.



```
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;
```

Figure 18-Generating a code Execution matrix based on RDTSC

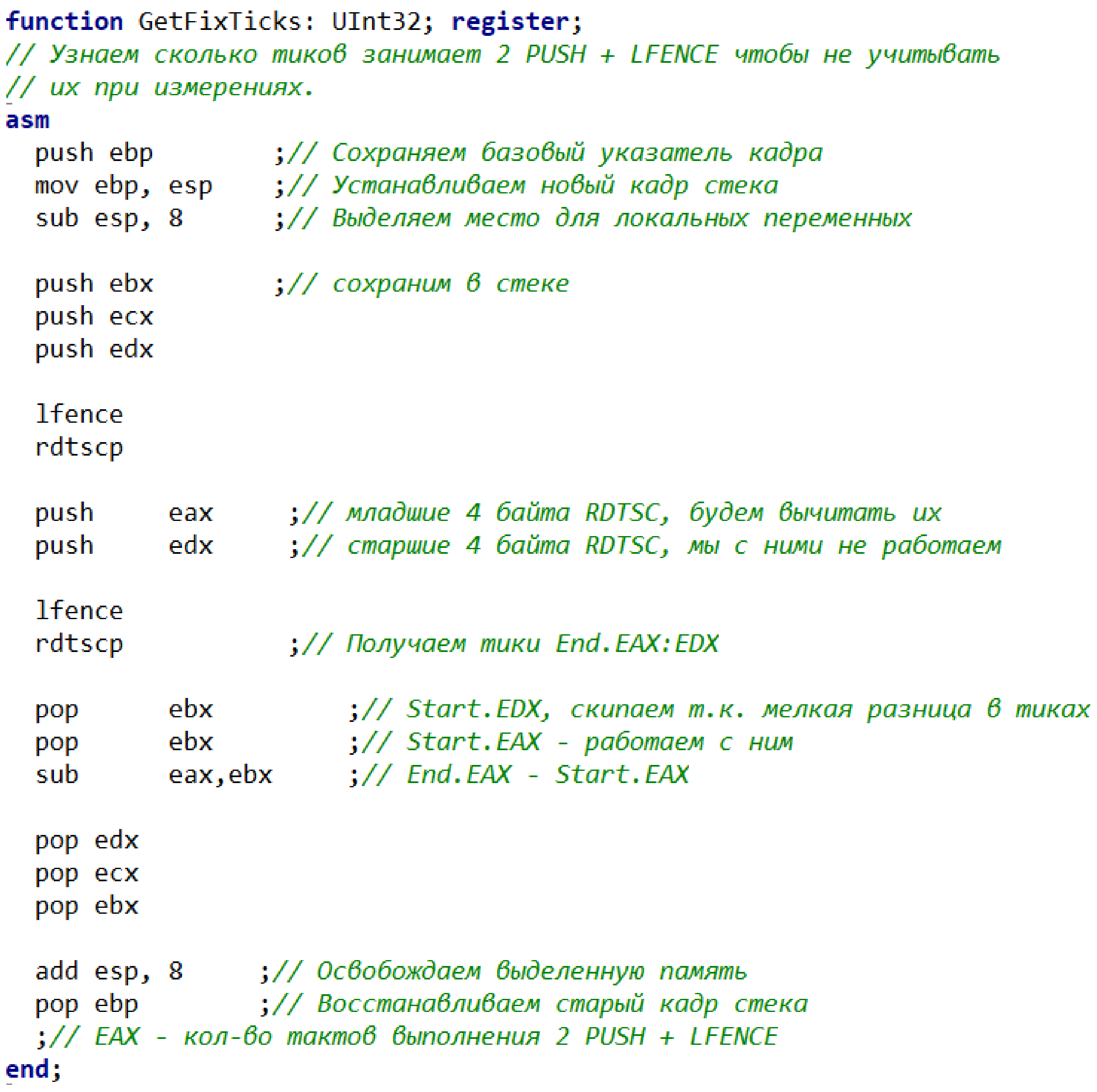
To get the CPU frequency in MHz, we measure the execution time of the WinAPI Sleep() function with a parameter of 500 ms, as shown in Figure 19. The received frequency is the maximum frequency at which the processor can operate. EndProfileRDTSC.inc stores two values – the number of ticks and the time in microseconds spent executing the CryptGenRandom() function.



```
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;
```

Figure 19-Getting the CPU frequency

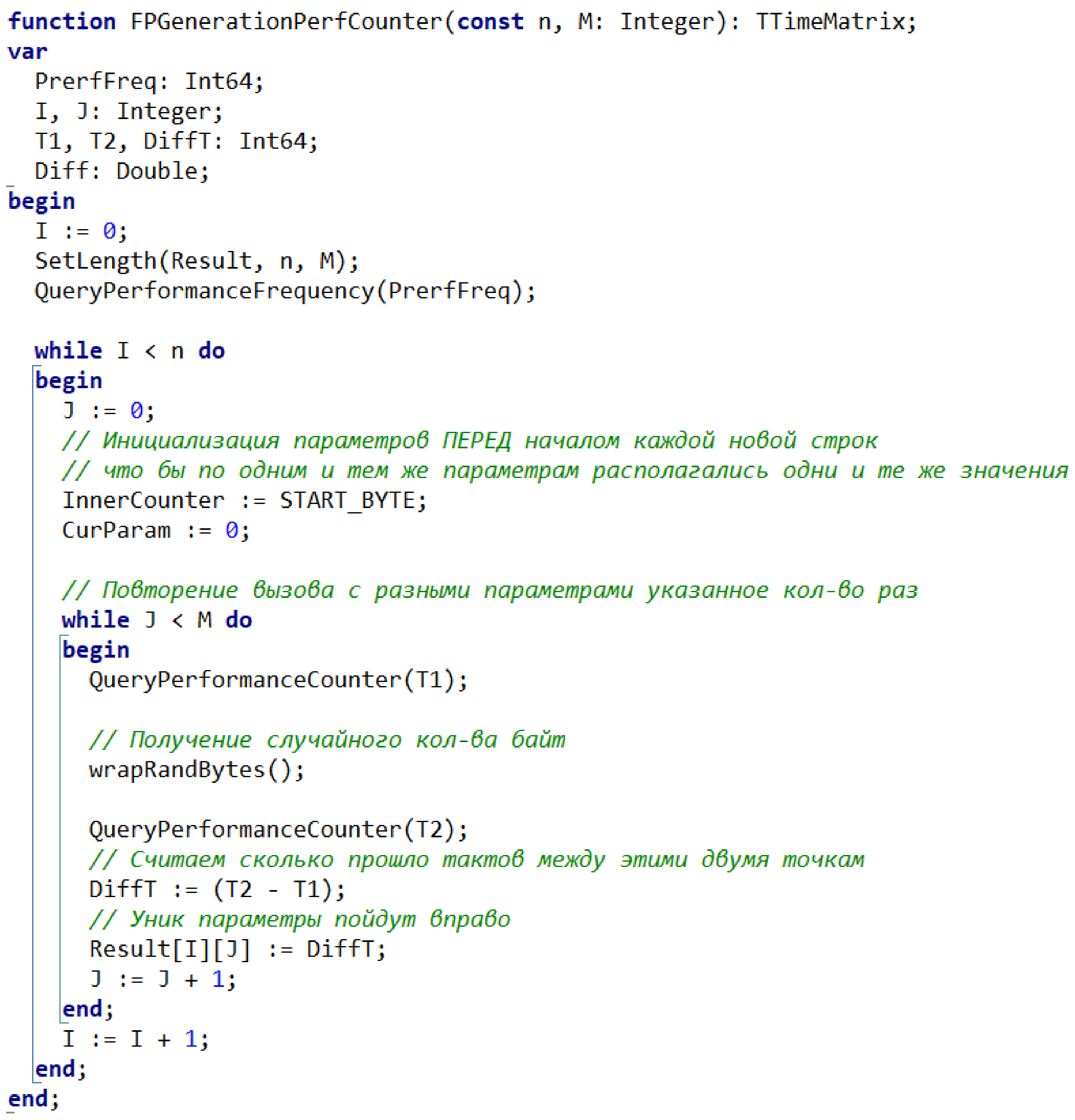
For more accurate measurements, we implemented a method for calculating the number of tick errors spent on commands participating in the measurement process, namely, the number of ticks occupied by the LFENCE command LFENCEand two PUSH commands [14], shown in Figure 20. However, subtracting the tick data in EndProfileRDTSC.inc did not affect the measurement accuracy.



```
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;
```

Figure 20-Getting the number of error ticks spent on RDTSC commands and working with the stack

After generating the matrix based on RDTSC, a second matrix is generated based on the system timer by calling WinAPI QueryPerfomanceCounter(), the generation code of which is shown in Figure 21.

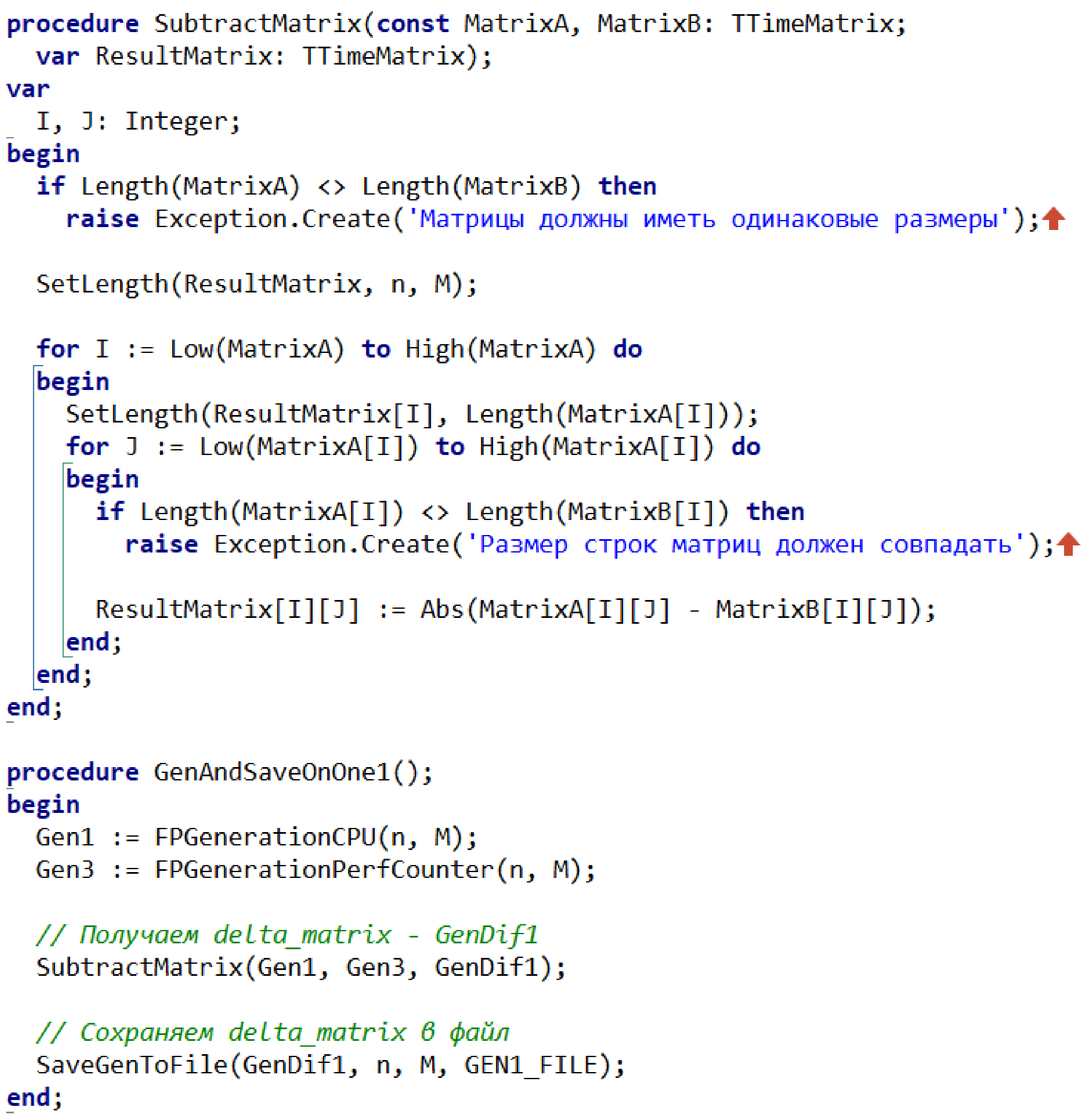


```
function FPGenerationPerfCounter(const n, M: Integer): TTimeMatrix;
var
  PrerfFreq: Int64;
  I, J: Integer;
  T1, T2, DiffT: Int64;
  Diff: Double;
begin
  I := 0;
  SetLength(Result, n, M);
  QueryPerformanceFrequency(PrerfFreq);

  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;
```

Figure 21 - Generating a Code Execution Matrix based on WinAPI QueryPerfomanceCounter()

The WinAPI QueryPerfomanceCounter() function returns the number of ticks of the system timer. To get the time, divide the number of ticks by the system timer frequency obtained by calling WinAPI QueryPerformanceFrequency().

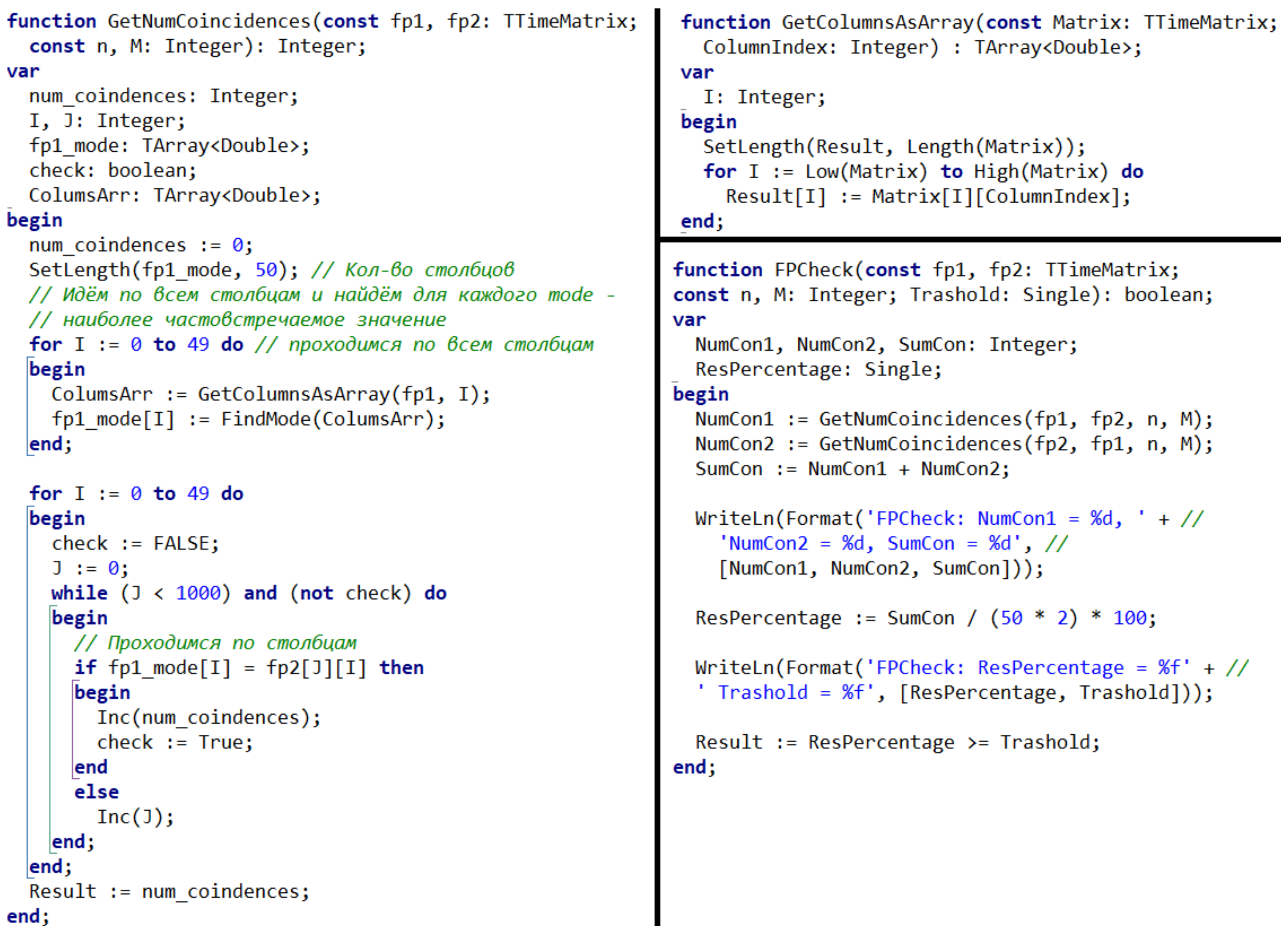
After generating two matrices, they are subtracted modulo each other, and the resulting delta matrix is considered a matrix of the code execution time of the current computer. The logic of matrix subtraction is shown in Figure 22.



```
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;
```

Figure 22-Generating two Code Execution matrices and finding the Delta matrix

After that, the resulting delta matrix is written to a file. On the second computer, exactly the same operations are performed, and then a comparison is made between the matrix of the first and second PCs, as shown in Figure 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;
```

Figure 23 - Finding mode in matrices and comparing matrices with each other

It is worth noting that the parameters for the measured CryptGentRandomCryptGentRandom() method are arranged in the same order for both matrices, as shown in Figure 24.

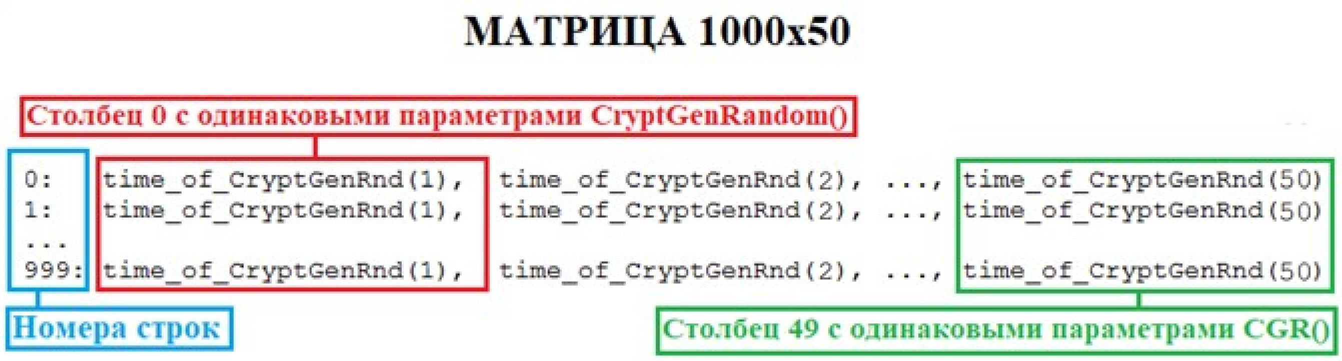
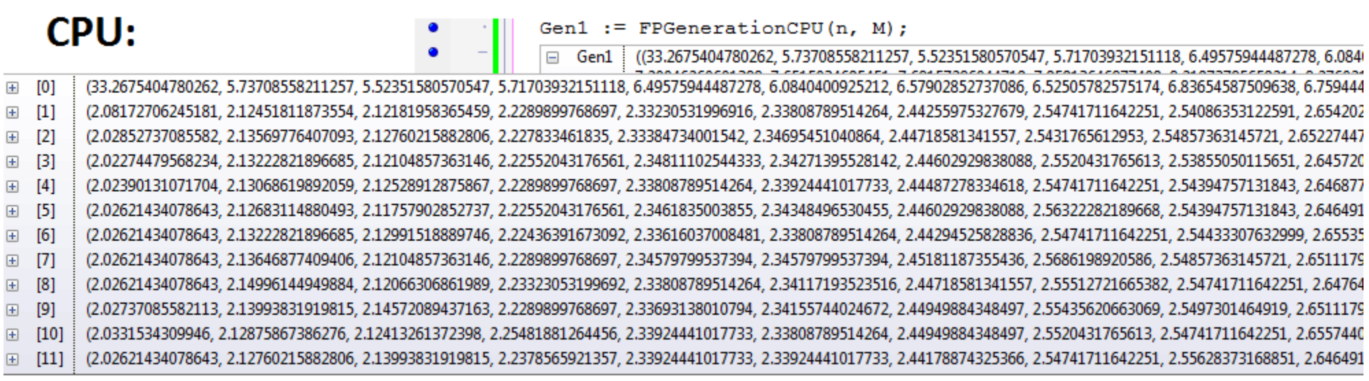

МАТРИЦА 1000x50
Столбец 0 с одинаковыми параметрами CryptGenRandom()
0: time_of_CryptGenRnd(1), time_of_CryptGenRnd(2), ..., time_of_CryptGenRnd(50)
1: time_of_CryptGenRnd(1), time_of_CryptGenRnd(2), ..., time_of_CryptGenRnd(50)
...
999: time_of_CryptGenRnd(1), time_of_CryptGenRnd(2), ..., time_of_CryptGenRnd(50)
Номера строк
Столбец 49 с одинаковыми параметрами CGR()

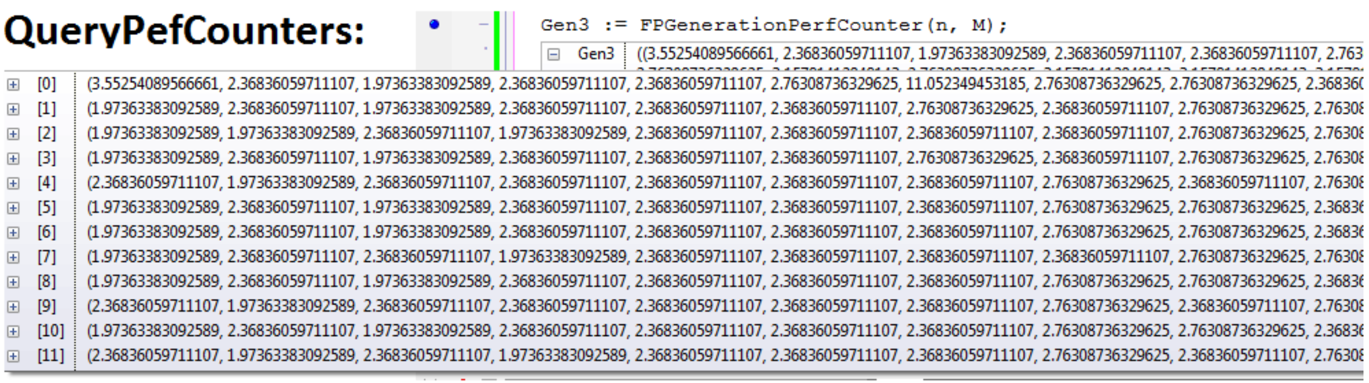
Figure 24-Matrix formation scheme taking into account input parameters for the measured CryptGenRandom()function

However, the same parameter for the measured function is not called twice in a row. This is done to improve the accuracy of the measurement, because the processor can cache the result of the first execution of the measured function during code execution and output it much faster when the function is called again with the previous input parameter. Thus, first the execution time of the CryptGenRandom() function is measured with parameter 10, then with parameter 11, and so on up to parameter 49 – At this time, the first row of the matrix is filled in. As soon as we reach the 49th parameter, we switch to the second row of the matrix and measure the function with parameter 10 again, etc.



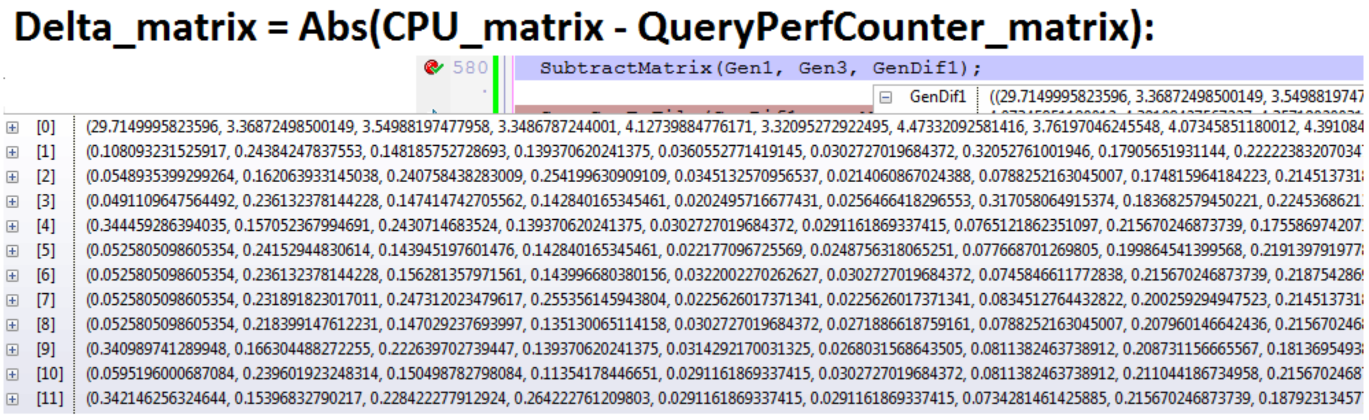
CPU:

```
Gen1 := FPGenerationCPU(n, M);
Gen1 | ((33.2675404780262, 5.73708558211257, 5.52351580570547, 5.71703932151118, 6.49575944487278, 6.084
[0]  (33.2675404780262, 5.73708558211257, 5.52351580570547, 5.71703932151118, 6.49575944487278, 6.0840400925212, 6.57902852737086, 6.52505782575174, 6.83654587509638, 6.759444
[1]  (2.08172706245181, 2.12451811873554, 2.12181958365459, 2.2289899768697, 2.33230531996916, 2.33808789514264, 2.44255975327679, 2.54741711642251, 2.54086353122591, 2.654202
[2]  (2.02852737085582, 2.13569776407093, 2.12760215882806, 2.227833461835, 2.33384734001542, 2.34695451040864, 2.44718581341557, 2.5431765612953, 2.54857363145721, 2.65227447
[3]  (2.02274479568234, 2.13222821896685, 2.12104857363146, 2.22552043176561, 2.34811102544333, 2.34271395528142, 2.44602929838088, 2.5520431765613, 2.53855050115651, 2.645720
[4]  (2.02390131071704, 2.13068619892059, 2.12528912875867, 2.2289899768697, 2.33808789514264, 2.33924441017733, 2.44487278334618, 2.54741711642251, 2.54394757131843, 2.646877
[5]  (2.02621434078643, 2.12683114880493, 2.11757902852737, 2.22552043176561, 2.3461835003855, 2.34348496530455, 2.44602929838088, 2.56322282189668, 2.54394757131843, 2.646491
[6]  (2.02621434078643, 2.13222821896685, 2.12991518889746, 2.22436391673092, 2.33616037008481, 2.33808789514264, 2.44294525828836, 2.54741711642251, 2.54433307632999, 2.65535
[7]  (2.02621434078643, 2.13646877409406, 2.12104857363146, 2.2289899768697, 2.34579799537394, 2.34579799537394, 2.45181187355436, 2.5686198920586, 2.54857363145721, 2.6511179
[8]  (2.02621434078643, 2.14996144949884, 2.12066306861989, 2.23323053199692, 2.33808789514264, 2.34117193523516, 2.44718581341557, 2.55512721665382, 2.54741711642251, 2.64764
[9]  (2.02737085582113, 2.13993831919815, 2.14572089437163, 2.2289899768697, 2.33693138010794, 2.34155744024672, 2.44949884348497, 2.55435620663069, 2.5497301464919, 2.6511179
[10] (2.0331534309946, 2.12875867386276, 2.12413261372398, 2.25481881264456, 2.33924441017733, 2.33808789514264, 2.44949884348497, 2.5520431765613, 2.54741711642251, 2.6557440
[11] (2.02621434078643, 2.12760215882806, 2.13993831919815, 2.2378565921357, 2.33924441017733, 2.33924441017733, 2.44178874325366, 2.54741711642251, 2.55628373168851, 2.646491
```



QueryPefCounters:

```
Gen3 := FPGenerationPerfCounter(n, M);
Gen3 | ((3.55254089566661, 2.36836059711107, 1.97363383092589, 2.36836059711107, 2.36836059711107, 2.763
[0]  (3.55254089566661, 2.36836059711107, 1.97363383092589, 2.36836059711107, 2.36836059711107, 2.76308736329625, 11.052349453185, 2.76308736329625, 2.76308736329625, 2.36836
[1]  (1.97363383092589, 2.36836059711107, 1.97363383092589, 2.36836059711107, 2.36836059711107, 2.36836059711107, 2.76308736329625, 2.36836059711107, 2.76308736329625, 2.76308
[2]  (1.97363383092589, 1.97363383092589, 2.36836059711107, 1.97363383092589, 2.36836059711107, 2.36836059711107, 2.36836059711107, 2.36836059711107, 2.76308736329625, 2.76308
[3]  (1.97363383092589, 2.36836059711107, 1.97363383092589, 2.36836059711107, 2.36836059711107, 2.36836059711107, 2.76308736329625, 2.36836059711107, 2.76308736329625, 2.76308
[4]  (2.36836059711107, 1.97363383092589, 2.36836059711107, 2.36836059711107, 2.36836059711107, 2.36836059711107, 2.36836059711107, 2.76308736329625, 2.36836059711107, 2.76308
[5]  (1.97363383092589, 2.36836059711107, 1.97363383092589, 2.36836059711107, 2.36836059711107, 2.36836059711107, 2.36836059711107, 2.76308736329625, 2.76308736329625, 2.36836
[6]  (1.97363383092589, 2.36836059711107, 1.97363383092589, 2.36836059711107, 2.36836059711107, 2.36836059711107, 2.36836059711107, 2.76308736329625, 2.76308736329625, 2.36836
[7]  (1.97363383092589, 2.36836059711107, 2.36836059711107, 1.97363383092589, 2.36836059711107, 2.36836059711107, 2.36836059711107, 2.36836059711107, 2.76308736329625, 2.76308
[8]  (1.97363383092589, 2.36836059711107, 1.97363383092589, 2.36836059711107, 2.36836059711107, 2.36836059711107, 2.36836059711107, 2.76308736329625, 2.76308736329625, 2.36836
[9]  (2.36836059711107, 1.97363383092589, 2.36836059711107, 2.36836059711107, 2.36836059711107, 2.36836059711107, 2.36836059711107, 2.76308736329625, 2.36836059711107, 2.76308
[10] (1.97363383092589, 2.36836059711107, 1.97363383092589, 2.36836059711107, 2.36836059711107, 2.36836059711107, 2.36836059711107, 2.76308736329625, 2.76308736329625, 2.36836
[11] (2.36836059711107, 1.97363383092589, 2.36836059711107, 1.97363383092589, 2.36836059711107, 2.36836059711107, 2.36836059711107, 2.76308736329625, 2.36836059711107, 2.76308
```



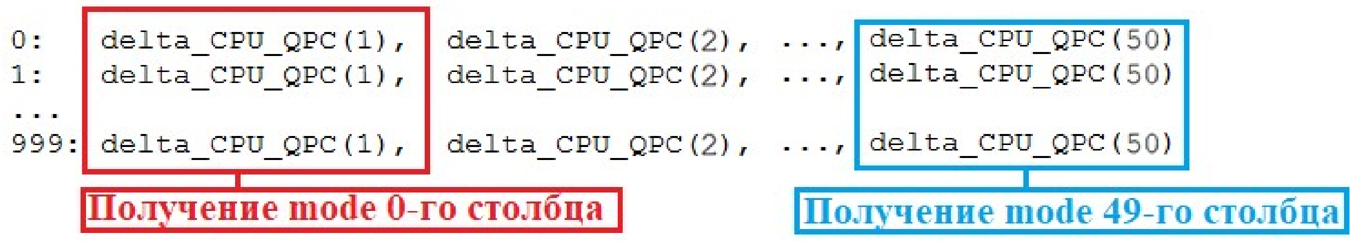
Delta_matrix = Abs(CPU_matrix - QueryPerfCounter_matrix):

```
580  SubtractMatrix(Gen1, Gen3, GenDif1);
GenDif1 | ((29.7149995823596, 3.36872498500149, 3.5498819747
[0]  (29.7149995823596, 3.36872498500149, 3.54988197477958, 3.3486787244001, 4.12739884776171, 3.32095272922495, 4.47332092581416, 3.76197046245548, 4.07345851180012, 4.391084
[1]  (0.108093231525917, 0.24384247837553, 0.148185752728693, 0.139370620241375, 0.0360552771419145, 0.0302727019684372, 0.32052761001946, 0.17905651931144, 0.22222383207034
[2]  (0.0548935399299264, 0.162063933145038, 0.240758438283009, 0.254199630909109, 0.0345132570956537, 0.0214060867024388, 0.0788252163045007, 0.174815964184223, 0.214513731
[3]  (0.0491109647564492, 0.236132378144228, 0.147414742705562, 0.142840165345461, 0.0202495716677431, 0.0256466418296553, 0.317058064915374, 0.183682579450221, 0.2245368621
[4]  (0.344459286394035, 0.157052367994691, 0.2430714683524, 0.139370620241375, 0.0302727019684372, 0.0291161869337415, 0.0765121862351097, 0.215670246873739, 0.175586974207
[5]  (0.0525805098605354, 0.24152944830614, 0.143945197601476, 0.142840165345461, 0.022177096725569, 0.0248756318065251, 0.077668701269805, 0.199864541399568, 0.21913979197
[6]  (0.0525805098605354, 0.236132378144228, 0.156281357971561, 0.143996680380156, 0.0322002270262627, 0.0302727019684372, 0.0745846611772838, 0.215670246873739, 0.218754286
[7]  (0.0525805098605354, 0.231891823017011, 0.247312023479617, 0.255356145943804, 0.0225626017371341, 0.0225626017371341, 0.0834512764432822, 0.200259294947523, 0.214513731
[8]  (0.0525805098605354, 0.218399147612231, 0.147029237693997, 0.135130065114158, 0.0302727019684372, 0.0271886618759161, 0.0788252163045007, 0.207960146642436, 0.215670246
[9]  (0.340989741289948, 0.166304488272255, 0.222639702739447, 0.139370620241375, 0.0314292170031325, 0.0268031568643505, 0.0811382463738912, 0.208731156665567, 0.181369549
[10] (0.0595196000687084, 0.239601923248314, 0.150498782798084, 0.11354178446651, 0.0291161869337415, 0.0302727019684372, 0.0811382463738912, 0.211044186734958, 0.2156702468
[11] (0.342146256324644, 0.15396832790217, 0.228422277912924, 0.264222761209803, 0.0291161869337415, 0.0291161869337415, 0.0734281461425885, 0.215670246873739, 0.18792313457
```

Figure 25-Example of generating two matrices and finding a delta matrix

Figure 25 shows the generated matrices using RDTSC and QueryPerfomanceCounter(), as well as the resulting delta matrix as a result of matrix subtraction modulo. Values are displayed in microseconds.

When comparing matrices, mode is calculated, which is the most frequently encountered value in a column with the same input parameters, as shown in Figure 26.



```
0:   delta_CPU_QPC(1),  delta_CPU_QPC(2),  ..., delta_CPU_QPC(50)
1:   delta_CPU_QPC(1),  delta_CPU_QPC(2),  ..., delta_CPU_QPC(50)
...
999: delta_CPU_QPC(1),  delta_CPU_QPC(2),  ..., delta_CPU_QPC(50)
```

Figure 26-Mode calculation scheme for delta matrix columns

After calculating mode for the first matrix, mode values are searched modein the same columns of the second matrix. If the mode value of the zero column of the first matrix was found in the zero column of the second matrix, then increase the matrix match counter and proceed to search for the mode of the next column. After processing the mode of the first matrix, mode is calculated for the columns of the second matrix and these modes are searched in the first matrix. At the end, the number of matches of both matrices is checked, and if the percentage of matches is greater than 50%, then both matrices are considered to belong to the same PC. [7]

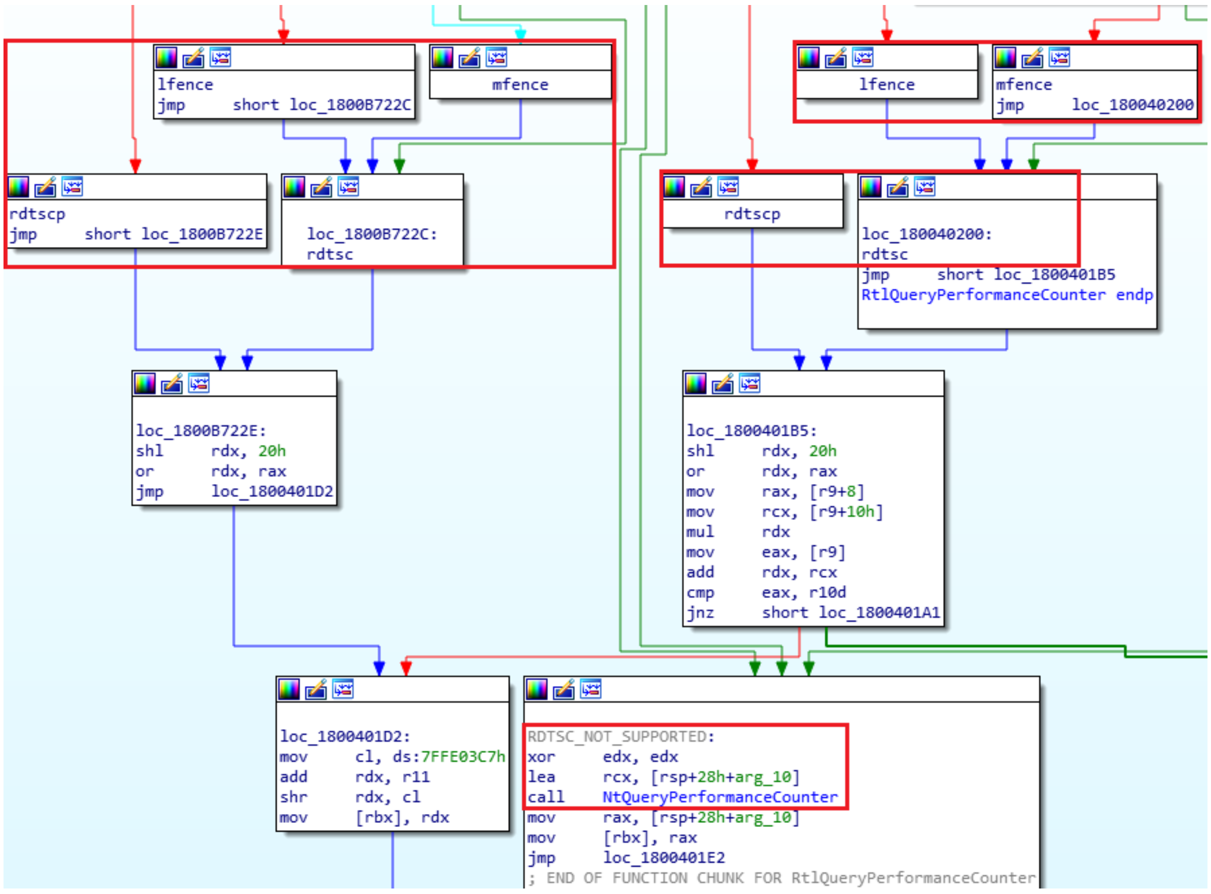
As a result, after numerous tests of generating delta matrices, there was still a lack of stability in determining whether the matrices belong to the same PC. It was found that the matrix assignment results change depending on the current processor load, which, according to Iskander's article, should not happen.

Stabilization of dynamic identification results

There Windowsare several different timers available for use on Windows:

1. RTC with a frequency of 32.768 kHz.
2. PIT with a frequency of 1.193180 MHz.
3. ACPI with a frequency of 3.58 MHz.
4. HPET with a frequency from 10 to 24 MHz.
5. LAPIC with no fixed frequency. [12]

If the processor supports an invariant iTSC tick counter in RDTSC, then QPCthe RDTSC or RDTSCP instructions will be called inside the QPC() functionRDTSC или RDTSCP, as shown in Figure 27. Moreover, the iTSC value in this case will be calibrated using HPET.



```
lfence
jmp     short loc_1800B722C

mfence

rdtscp
jmp     short loc_1800B722E

loc_1800B722C:
rdtsc

loc_1800B722E:
shl     rdx, 20h
or      rdx, rax
jmp     loc_1800401D2

lfence

mfence
jmp     loc_180040200

rdtscp

loc_180040200:
rdtsc
jmp     short loc_1800401B5
RtlQueryPerformanceCounter endp

loc_1800401B5:
shl     rdx, 20h
or      rdx, rax
mov     rax, [r9+8]
mov     rcx, [r9+10h]
mul     rdx
mov     eax, [r9]
add     rdx, rcx
cmp     eax, r10d
jnz     short loc_1800401A1

loc_1800401D2:
mov     cl, ds:7FFE03C7h
add     rdx, r11
shr     rdx, cl
mov     [rbx], rdx

RDTSC_NOT_SUPPORTED:
xor     edx, edx
lea     rcx, [rsp+28h+arg_10]
call    NtQueryPerformanceCounter
mov     rax, [rsp+28h+arg_10]
mov     [rbx], rax
jmp     loc_1800401E2
; END OF FUNCTION CHUNK FOR RtlQueryPerformanceCounter
```

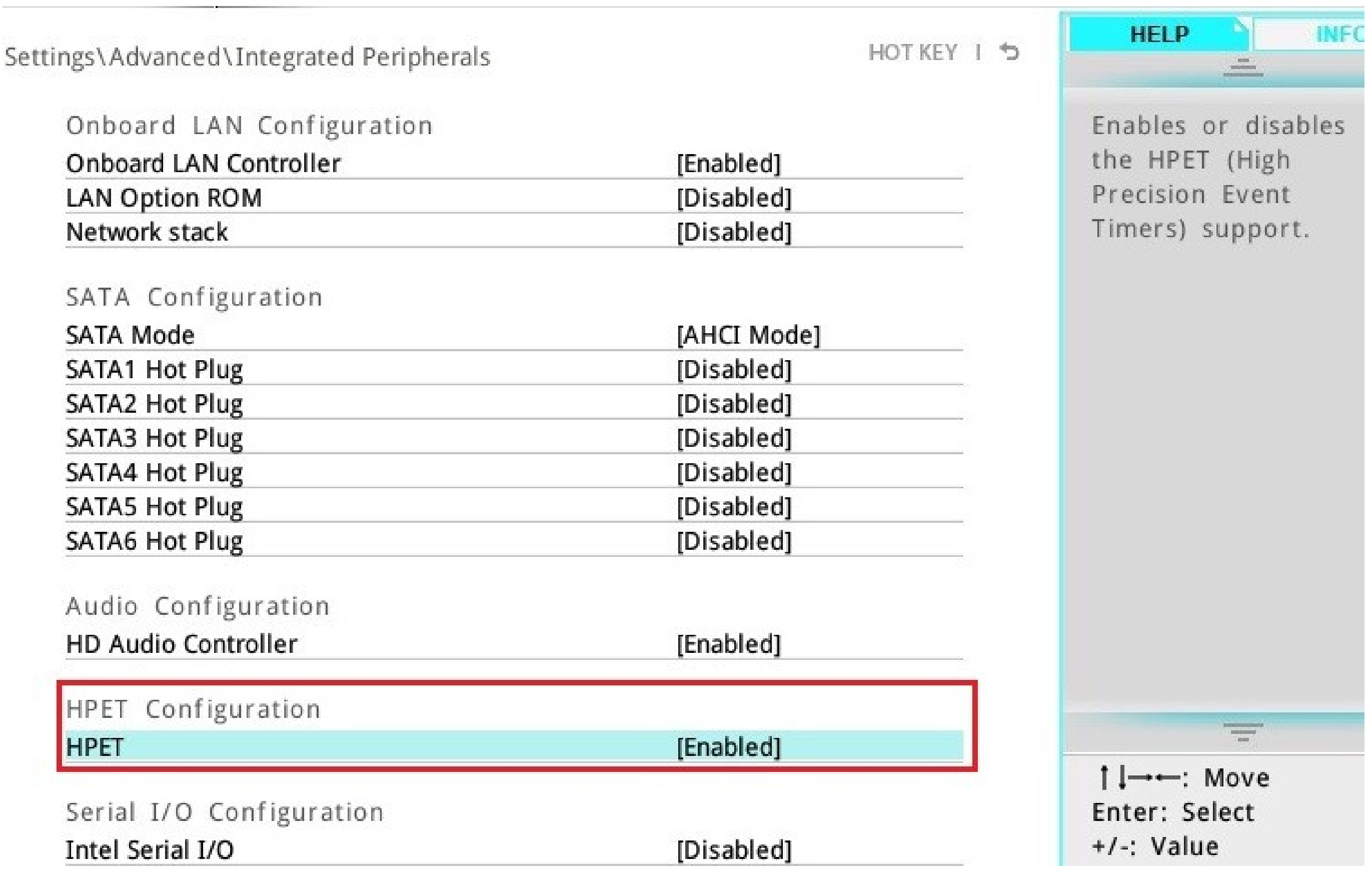
Figure 27 - Disassembled listing функциоf the ntdll.RtlQueryPerformanceCounter function

If iTSC is not supported by the processor, QPC() will call the kernel mode function NtQueryPerformanceCounter (), which will refer to one of the available hardware timers – HPET or ACPI.

According to the recommendations received from the authors of scientific articles Iskander and Miguel, it is necessary to use different timers with different frequencies to generate matrices. Currently, RDTSC and QPC() refer to the same iTSC counter. To switch QPC to HPET, run the following command in cmd.exe.exe as an administrator:

bcdedit /set useplatformclock true

and then restart the PC [17]. You should also make sure that UEFI/the HPET timer is enabled in the UEFI/BIOSHPET, which is configured in Figure 28.


Settings\Advanced\Integrated Peripherals

Onboard LAN Configuration

Onboard LAN Controller	[Enabled]
LAN Option ROM	[Disabled]
Network stack	[Disabled]

SATA Configuration

SATA Mode	[AHCI Mode]
SATA1 Hot Plug	[Disabled]
SATA2 Hot Plug	[Disabled]
SATA3 Hot Plug	[Disabled]
SATA4 Hot Plug	[Disabled]
SATA5 Hot Plug	[Disabled]
SATA6 Hot Plug	[Disabled]

Audio Configuration

HD Audio Controller	[Enabled]

HPET Configuration

HPET	[Enabled]

Serial I/O Configuration

Intel Serial I/O	[Disabled]

Enables or disables the HPET (High Precision Event Timers) support.

↑↓→←: Move
Enter: Select
+/-: Value

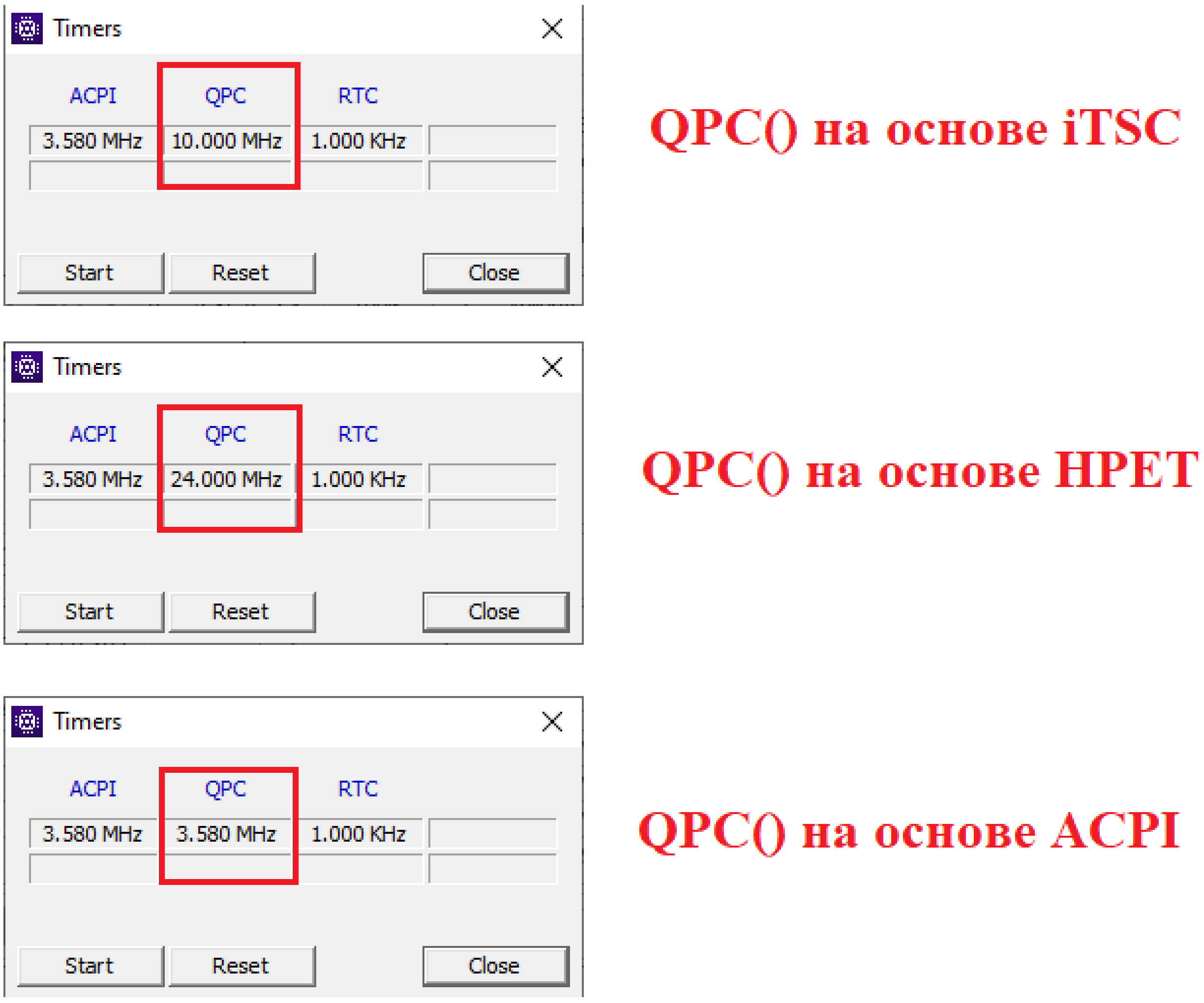
Figure 28-Configuring HPET Activation in UEFI/BIOS

If you run **the bcdedit useplatformclock** command with HPET disabled in UEFI/BIOS, QPC() will use the ACPI timer after the OS is rebooted. To return to using iTSC in QPC(), run the following command in cmd.exe.exe as an administrator:

bcdedit /set useplatformclock false

and then restart the PC.

To check whether the timer used in QPC() has changed, you need to query the system timer frequency via WinAPI QueryPerfomanceFrequency() or view the QPC frequencies in the "CPU-Z" utility [18], as shown in Figure 29.



Timers

ACPI	QPC	RTC
3.580 MHz	10.000 MHz	1.000 KHz

Timers

ACPI	QPC	RTC
3.580 MHz	24.000 MHz	1.000 KHz

Timers

ACPI	QPC	RTC
3.580 MHz	3.580 MHz	1.000 KHz

Figure 29-Differences in timer frequencies for QPC in the "CPU-Z"program

When using HPET or ACPI as the basis of QPC, we noticed a deterioration in the responsiveness of the OS. This is due to the time delays required to get the value of the HPET or ACPI timers, when both getting the iTSC value is almost instantaneous.

On Windows, a limited number of run-time measurement functions are available in user mode, referring to timers with different frequencies. The main sources for measuring code execution time are:

1. GetSystemTimeAdjustment () – returns the OS program timer interval of 15.625 msec with a frequency of 1/0.015625 = 64 Hz. Derivatives of a function with the same frequency:
   - GetTickCount()
   - GetSystemTime()
   - timeGetTime()

   These functions cannot be used to measure the speed of code execution, because they operate at too low a frequency.

2. QueryPerformanceFrequency () – uses iTSC with a virtual frequency calculated dynamically based on hardware capabilities (in this case, 10 MHz), as well as an HPET timer (from 10 to 24 MHz) or ACPI (3.59 MHz).
3. Ассемблерная инструкция The RDTSC or RDTSCP assembly instruction to get the iTSC.

Two matrices were calculated using QPC() based on the HPET timer and RDTSC based on iTSC. The test console application will be compiled under x32 to use assembler inserts with RDTSC.

Local testing on one PC was performed by generating two matrices, finding the delta matrix, and saving it to the Gen file1.txttxt next to the program. After that, there was a 5-second wait and the subsequent generation of two code execution matrices, the delta matrix was calculated and saved to a Ge file. 2.txttxt. Then Gen was loaded1.txttxt and Gen2.txttxt files and matrix comparison to determine whether both matrices belong to the same PC. This procedure was repeated 10 times in a row. As a result, local tests on one PC showed a 100% match of delta matrices in 10 cases out of 10 in the absence of any CPU load. However, if you generate a delta matrix without CPU load, and then generate a delta matrix with the processor stress test running simultaneously and compare both delta matrices, the result of checking the identity of the matrices can vary from 40% to 90% of matches, which is an unstable result. For CPU load stress tests, we used the "CPU Stress" utilityCPU and the "CPU Stress" functionStress CPUin the "CPU-Z" program.

In his article, Iskander described stable matrix generation even at 100% CPU load, which could not be repeated in the current work.

To improve the stability of matrix generation, the following attempts were made:

1. Replacing RDTSC with the more modern assembly instruction RDTSCP. We also tested replacing the LFENCE command with CPUID, MFENCE for waiting for pipeline execution/clearing the CPU command pipeline.
2. Compilation under x64 for direct WinAPI calls bypassing the WoW64 layer. When switching to x64, it was necessary to abandon the use of embedded assembler inserts, since xthey are prohibited in x64Delphi. Instead of inserts, the usual call to assembler time measurement functions is used.
3. Testing various variations of input parameters (generated bytes) of the CryptGenRandom() function, as well as changing the parameter step between calls.
4. Changing the measured function from CryptGenRadom() to FNV1Hash() and BobJenkinsHash() with pre-generated input values.
5. Improving the accuracy of QPC() by calling WinAPI timeBeginPeriod(1)
6. Calling WinAPI Sleep(1) before each measurement of code execution time, to reduce the impact of CPU load on the result. Using a zero argument did not affect the result, and using one as an argument significantly extended the matrix generation time and slightly stabilized the result.
7. Changing the approach to matrix generation, in which both matrices were filled at once, and not separately (after filling the zero row of the zero column of the RDTSC/P matrix, QPC() was measured and QPCthe zero row of the zero column of the QPC () matrix was filled).
8. Disabling UEFIthe following processor technologies in UEFI on a Windows 10 PC:
   - Hyper-threading
   - TurboBoost
   - DVFS
   - Enhanced Intel SpeedStep

However, it was not possible to achieve stabilization when comparing local delta matrices. Still, even after completing all the above points, a matrix generated without CPU load and with a load is considered two matrices that are not very similar (the percentage of identity of the two local matrices still varies about 70%), as shown in Figure 30.

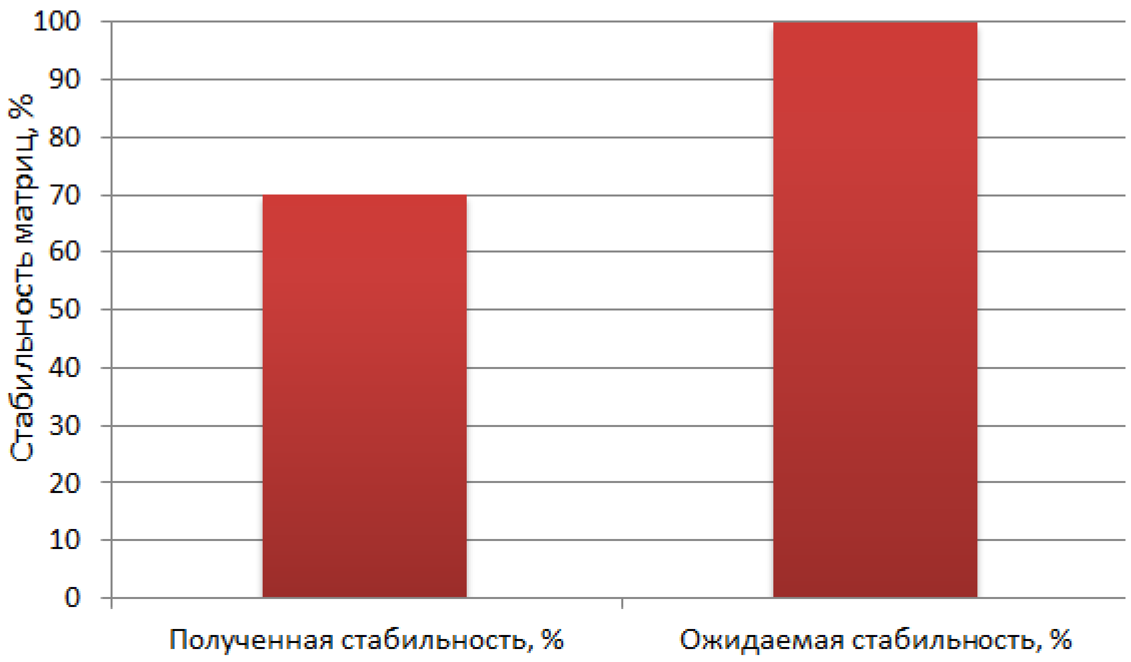


Figure 30-Comparison of the obtained and expected stability of delta matrices

It is worth noting that generation testing and matrix comparison were performed only on two PCs with different hardware and software tools. Moreover, the Windows 7 PC was a laptop with the BIOS blocked by the manufacturer (there was no possibility to configure the processor, RAM, etc.).

Recommendations for further research

Despite the fact that in this paper it was not possible to implement a stable generation of code execution rate matrices for dynamic PC identification, it is recommended to continue trying to implement the described mechanism, taking into account the experience of current work, namely:

1. You must perform generation testing on multiple PCs with the same hardware and software components. This was not done in this paper, because it was not possible to obtain permission from the university to experiment with educational PCs.

2. Work with HPET and ACPI timers directly from kernel mode, not limited to user mode. In this work, only the WinAPI user mode functions were used.
3. Test HPET and ACPI matrix generation ACPIfrom the OS kernel without using RDTSC/RDTSCP.
4. Stabilize the frequency of RAM operation by making settings in the BIOS/UEFI.
5. Try using STM to avoid interrupting the OS scheduler when working with memory.
6. Perform tests on various Intel AMD processors. There is a possibility that this approach may work stably only on certain processor models/series. However, Iskander could not remember on which PC processors his testing was performed.

List of sources used

1. Sanchez-Rola, I. Santos, D. Balzarotti, Clock Around the Clock: Time-Based Device Fingerprinting [Electronic resource] URL: https://www.s3.eurecom.fr / docs/ccs18_iskander. pdf (accessed: 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 [Electronic resource] URL: https://www.sciencedirect.com/science/article/pii/S108480452200220X (accessed: 16.04.2025)

3. Btbd, HWID spoofer for Windows [Electronic resource] URL: https://github.com/btbd/hwid (accessed: 20.04.2025)

4. SamuelTulach, Manual map kernel mode driver [Electronic resource] URL: https://github.com/SamuelTulach/nullmap (accessed: 20.04.2025)

5. Samuel Tulach, Windows kernel-mode hardware identifier (HWID) spoofer [Electronic resource] URL: https://github.com/SamuelTulach/mutante (accessed: 20.04.2025)

6. Marylin, System timers. Part[6] - Fundamentals of code profiling [Electronic resource] URL: https://codeby.net/threads/sistemnyye-taimery-chast-6-osnovy-profilirovaniya-koda.74153/ (accessed: 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 [Electronic resource] URL:https://www.cs.usfca.edu/~cruse/cs210s06/rdtscpm1-1.pdf (accessed: 16.04.2025)

9. MSDN BCDEdit /set [Electronic resource] URL: https://learn.microsoft.com/ru-ru/windows-hardware/drivers/devtest/bcdedit--set://learn.microsoft.com/ru-ru/windows-hardware/drivers/devtest/bcdedit--set (accessed on 01.05.2025).

10. CPU-Z [Electronic resource] URL: https://www.cpuid.com/softwares/cpu-z.html://www.cpuid.com/softwares/cpu-z.html (accessed on 01.05.2025).

11. RW Utility [Electronic resource] URL: https://rweverything.com://rweverything.com (accessed on 01.05.2025).