RC5 Benchmark on Raspberry Pi 3 B+

Abraham Nkum, Arron Lu, Ashley McBean, **Mohamed El-Hadedy**

College of Electrical and Computer Engineering

California State Polytechnic University, Pomona

Pomona, USA

{arronlu, annkum, amcbean, **mealy**}@cpp.edu

*Abstract*—The purpose of this project is to benchmark the performance of the RC5 encryption algorithm on a Raspberry Pi. The specific model chosen was the Raspberry Pi 3 B+ running Ubuntu 20.04 and compared with a laptop running Ubuntu 20.04. The execution of each RC5 algorithm was run 100 times and the averages were obtained and showed that the RC5 algorithm included in Ronald Rivest’s original paper, though it is less functional than the Apple Open Source version.

Keywords—RC5, Raspberry Pi 3, Raspberry Pi 3 B+

# Introduction

Many modern-day security measures, such as RSA, derive their cryptographic strength from prime number factorization. This has allowed security measures to be very secure for the last few decades, but advances in the computation world are threatening that security. Shor’s algorithm, in particular, will allow quantum computers to consistently break public key algorithms that derive their cryptographic strength from prime number factorization. To combat the security issues that quantum computing brings, other security protocols that do not depend on prime number factorization have to be developed. RSA is an outdated security protocol that is not secure at low round numbers and has had 56- and 64-bit keys broken. The purpose of this project is not to update RC5 to a quantum resistant security algorithm, but to explore the capabilities of an algorithm that derives its cryptographic strength from shifting on a Raspberry Pi 3. Doing so will give a better idea of whether a shift-based algorithm, in its simplest form, is applicable for future applications given its resource consumption.

# Related Works

Two other papers published in IEEE database also explore the capabilities of RC5. [3] tests the RC5 algorithm along with CAST on a laptop. [3] makes no mention as to what implementation were tested. It is assumed that their version of RC5 would be a software version, as they are testing on a laptop. [4] tests the RC5 algorithm using a hardware implementation on a Virtex II Xilinx FPGA. The language used was VHDL. In regards to RC5, [3] tested RC5 across millions of cycles to see how the byte transmission speed of RC5. [3] also had a power measurement, but marked the power in terms with the remaining battery life of a laptop. This is an incredibly poor measure of power due to the numerous ways that a laptop’s battery life can be changed from the original and none of which are mentioned. [4] tests the speed of the RC5 hardware implementation very robustly but leaves out power. This work aims to benchmark RC5 on a Raspberry Pi 3 B+ while simultaneously benchmarking the speed and power usage. The scope of the speed benchmarking will not be as wide as [3] but should be wide enough to properly gauge RC5’s speed capabilities while the power analysis will give a good idea of how useful RC5 can be for more limited systems.

# RC5’s Purpose

The basis of RC5 is rooted in speed, adaptability, and simplicity [1]. For simplicity, RC5 was designed to use very little memory and intended for software and hardware applications. Regarding adaptability, RC5 was designed to have a variable amount of security depending on certain adjustable parameters, particularly a variable secret key length and number of rounds. RC5 was also designed for use in 32-bit systems but also could function in 64-bit systems. A very slight addition is needed to ensure 64-bit functionality but might not be necessary depending on the processor used. With those qualities, RC5 was very suitable for use in constrained environments, such as Bluetooth devices and RFID devices, which begs the question why is RC5 not being used nowadays. The quick and simple answer is that RC5 is just not secure at low round numbers, 18 to 20 rounds are suggested for sufficient protection, and many of the messages encrypted with high bit secret keys have been broken already [5]. The obvious way to combat these problems would be to increase the number of rounds used and the number of bits that the secret key contained but doing so would compromise many of the benefits that RC5 bring to constrained environments. Combined with the fact that other encryption algorithms were more secure with less drawbacks, RC5 was thrown to the wayside. Nonetheless, RC5 is still a very simple and easy to implement encryption algorithm and this project will study a couple of its software implementations.

# RC5 Basics

## The Operations

To keep RC5 simple and easily implementable on software and hardware systems, RC5 only uses basic, or primitive, operations. These operations are found on most processors which is what allows RC5 an extra degree of versatility fir software designs. Hardware designs can also take advantage of the RC5 operations also, as all the operations can be easily synthesizable in Verilog or VHDL. The operations are the following: 2’s complement addition, 2’s complement subtraction, bitwise exclusive OR, left rotation, and right rotation. Although there are 5 operations listed, 2’s complement addition and 2’s complement subtraction are inverses of each other, same with left rotation and right rotation, so there are only 3 “unique” operations employed by RC5. Note that left and right rotation are a bit more complex in their definition. While regular left and right rotation might be defined as follows: x >>> y, the RC5 left and right rotation have y interpreted as a little more than just a number. y is also modulo w. For this specific project, w is defined as 32 so, effectively, only the last 5 bits of y will be used to determine the shift amount instead of the entirety of y. For simplicity, this version of left and right rotation will just be referred to as left and right rotation but keep in mind that the RC5 specific version of left and right rotation is being referred to.

## Adjustable Parameters

In the original paper from Ronald Rivest detailing the different parameters, there are 3 different variables that can be adjusted. These 3 variables are word size, rounds, and bytes in the secret key.

Word size refers to how big each word is in bits. Each word contains w/8 bytes will influence how many bits the cyphertext, plaintext, and the secret key will be. Increasing the word size will increase the difficulty in breaking the cyphertext but will require more memory to be used for storing the cyphertext and the plaintext. Note that the plaintext could be initialized with a size smaller than the word size to save some memory if only a larger size cyphertext were desired. In the most general form of the code, provided in Ronald Rivest’s paper, initializes the plaintext variable as type word. This makes the code more versatile by allowing for changes in the word size to be reflected easily throughout the entire code but would use more memory if the entire capacity of the plaintext is not utilized. Nevertheless, an increase in the number of bytes per word would increase the overall memory required by the algorithm quite a bit.

The number of rounds refers to the number of times that the plaintext has a set of operations performed on it. This will be discussed in more detail later, but for now accept that the greater the number of rounds the more complex the cyphertext will be to decrypt. From just the information provided, increasing the number of rounds only increases the computation time but because the number of rounds also determines the size of the expanded secret key table. The expanded secret key table basically refers to the number of secret keys needed to perform each round. Each round requires a different secret key created during secret key expansion so there is no way around this drawback. As a result, an increase in the number of rounds would increase the computation time required for all three processes: encryption, decryption, and expanding the secret key, as well as increasing the memory required by the secret keys.

The number of bytes for each secret key can also be adjusted for RC5 from 0 to 255. Increasing the number of bytes each secret key has will increase the security but will also heavily increase the amount of memory that the algorithm will need. Increasing the word size, rounds, and bytes in the secret key will increase the security that RC5 provides but at the tradeoff of memory used and time needed for the various processes. While this is an intended tradeoff for RC5, massively increasing the different word parameters would counteract many of the benefits that RC5 brings. Given that RC5 has already been proven to be vulnerable to differential cryptanalysis with known plaintext and cyphertext pairs at low round numbers, a high number of rounds must be utilized to insure a basic amount of security. Higher bit values for words and the secret key are also suggested for a higher level of security but increasing the number of rounds is mandatory for a basic level of security in the modern era.

For this project, only the suggested parameters are suggested which are a word size of 32, 12 rounds, and a 10-byte secret key are benchmarked. Ronald Rivest abbreviates the parameters of RC5 with the following: RC5 – w/r/b. For example, the suggested parameters would then be denoted RC5 – 32/12/10.

## Setting or Expanding the Secret Key

RC5 is split up into three different processes: encryption, decryption, and setting the secret key. The first process that much occur is setting the secret key. Setting the secret key requires an input of a key value. In the code that Ronald Rivest gives as an example of RC5, the key is set as an unsigned char value. This value can be adjusted to different types, but the most important part is that the input key is the same number of bits that is specified in the code. In the first few lines of Ronald Rivest’s example code, the number of bits that a RC5 key is designated. The value itself plays no role in the code beyond a loop in the secret key setting process and can be hard coded to the number of bytes in the key instead. Nonetheless, the key needs to be the appropriate number of bytes specified in the secret key setting process, but the type of the key does not matter. From there, setting the secret key can be split up into four different parts: defining the magic constants, converting the secret key from bytes to words, initializing array S, and mixing in the secret key in that order.

The magic constants needed for RC5 are referred to as and , where w refers to the word size parameter for both and . is defined with the following equation: where e is the base of natural logarithms. is defined as the following equation: where ϕ is the golden ratio. Both and can be any value but this project only benchmarks the suggested parameters of RC5 – 32/16/10 which yields a value of b7e15163 and a of 9e3779b9. Note that both and are hexadecimal representations of the number that is calculated from and . Given that this project only benchmarks the suggested parameters, finding and was already done so this step was, effectively, skipped. This process is better done by hand or calculated separately. Calculating the magic numbers real time with different word values is not suggested and could yield errors due to the irrational nature of e and ϕ. The determining of the magic key is therefore not referred to as an algorithmic step in expanding or setting the secret key but is an important step regardless.

After finding and , the next step in setting the secret key is to convert the secret key from bytes to words. Another array called L of size c, where c is b/u, is then initialized. u is defined as w/8 which is basically the number of bytes that is in a word. So, another way to think about c is that c is basically the total number of bytes that make up the secret key size divided by the number of bytes that make up a word which is the number of words needed to make up the secret key. In formula form, . In other words, a new array called L with size equal to the number of words needed to makeup the key is initialized. The values in the secret key are then copied over from the secret key into L with the following loop in the given code:

for (i=b-1,L[c-1]=0; i!=-1; i--) L[i/u] = (L[i/u]<<8)+K[i];

This code basically takes the most significant values of the secret key, adds that value to L, then right shifts that value by 8 and repeats that process until there are no more values left in the key. In the case that b and c are equal to 0, c is reset to 1 and L[0] is set to 0.

The next step is to then initialize the secret key table or the expanded secret key, referred to as S. S is size t where t is determined by 2\*(r+1) where r is the number of rounds. The first value of S is initialized to and every other value is made to be . The following code given in Ronald Rivest’s paper performs the aforementioned process:

for (S[0]=P,i=1; i<t; i++) S[i] = S[i-1]+Q;

Lastly, the input secret key is mixed into S to finish expanding or setting the expanded secret key. This step is probably the most intense step of the secret key in that this step has the most complex procedure of all the steps. The discussion of this complex process starts with two words. For the most part, the two words act like temporary variables for this step and are called A and B in the given code, with initial values of 0. Another two variables, i and j, are used for indexing purposes. i is defined as (i+1) mod(t) and j is defined as (j+1) mod(c). In equation form, i=(i+1) % t and j=(j+1) % t. Both i and j are initialized at 0. A is then made equal to S at index i, referred to as S[i] for simplicity, and both A and S[i] are made equal to S[i] added to A, B, and then right shifted by 3. In equation form, A = S[i] = (S[i] + A + B) <<< 3. B is made equal to L at index j, L[j], and both B and L[j] are made equal to L[j] added A, B, and right shifted by A plus B. In equation form, B = L[j] = (L[j] + A + B) <<< (A+B). This whole process is then repeated 3 multiplied by t, the size of the expanded secret key, and c, the number of words in the key. The following given code accomplishes the aforementioned mixing process:

for (A=B=i=j=k=0; k<3\*t; k++,i=(i+1)%t,j=(j+1)%c)

{

A = S[i] = ROTL(S[i]+(A+B),3);

B = L[j] = ROTL(L[j]+(A+B),(A+B));

}

Note that 3\*t is used instead of the maximum of t and c. This is because t is usually larger than c, so t can be used as a in place of the max of t and c. With that, setting and expanding the secret key is done which means encryption and decryption can be discussed.

## Encryption and Decryption

Encryption and decryption, ironically, is much simpler than key expansion. Since decryption is basically the inverse of encryption, encryption will be discussed in depth and decryption will basically just be identifying how to reverse the encryption process.

Encryption assumes that the inputs and output will be 2 words. The first step to encryption is to then assign two temporary words, called A and B. A is made equal to the first word of the input and B is made equal to the second word of the input. A is then added to the first value, or the zeroth value, of the expanded secret key and B is added to the second value of the expanded secret key. An indexing variable, called i, is also used to index the appropriate expanded secret key value. i is incremented by 1 every loop, starts at 1, and ends at r where r is the number of rounds. A is then XORed with B, left rotated by B, and added to the expanded secret key value at index 2\*i. B is then XORed with A, left rotated by A, and added to the expanded secret key value at (2\*i) + 1. This whole process is then repeated r-1 times. Finally, the lower word of the output is made equal to A and the higher word of the output is made equal to B. Note that the input for encryption will the plaintext and the output will be cyphertext.

As previously mentioned, decryption for RC5 is simply the inverse of encryption. Therefore, decryption just needs to invert the process of encryption. Decryption still has an input and output of two words, so A and B are still utilized. A is the first word of the two-word cyphertext input and B is the second word of the cyphertext input. Decryption still also needs to access different values of the expanded secret key, so an indexing variable i is still needed. Whereas encryption’s last step was adding B to the expanded secret key value at index 2\*i + 1, decryption will do the opposite so decryption’s first step will be to take B and subtract the expanded secret key value at index 2\*i + 1. B is then right rotated by A and finally XORed with A. For A, A is subtracted by the expanded secret key at index 2\*i. That value is then right rotated by B and the result is XORed with B. This process is repeated r-1, where r is the number of rounds, times. Then A is subtracted by the expanded secret key value at index 0 and B is subtracted by the expanded secret key value at index 1. The 2-word plaintext message is then composed of A and B with A being the first word and B being the second word.

# Implementations

In this project, there are 2 different implementations of the RC5 algorithm. Both implementations are in software. A hardware implementation of RC5, done in Verilog is discussed.

## Ronald Rivest Implementation

In the original paper detailing RC5, Ronald Rivest provides an implementation of RC5. This implementation is the most versatile version available as this implementation is the most parameterized out of all the implementations. This implementation is also the one referenced during the description of the encryption, decryption, and key expanding processes. Regarding improvements with memory or execution time, there is not much to be done. The Ronald Rivest implementation is incredibly simple and most of the changes would not be with the implementation of RC5, but obscure interactions with compilers or very minor adjustments. One interesting concept that Ronald Rivest uses is pass by reference, though the way it is utilized is quite interesting. For the three different functions core to RC5, all the functions made take inputs of pointers to type word. Word is simply defined as an unsigned long int in the code, but any variable type with the appropriate number of bits that can be left and right rotated will suffice. Those values are then copied in each of the functions and then manipulated until the appropriate result is obtained. One big coding concept in C and C++ is the language is the difference between pass by reference and pass by value. Pass by value is when the actual value that a variable holds is copied to a temporary variable and then the variable that is being passed to gets a copy of the value. Pass by reference is when the address of the variable is passed instead, so the variable that is being passed to only sees the actual value of the original variable. Basically, pass by value will result in two variables containing the same information to exist while pass by reference will have one variable containing information and another variable that knows where the information is. Pass by value is superior to pass by reference regarding memory and speed in that way and, in most cases, should be used. For the most part though, the way that the different functions operate makes the difference pass by reference and pass by value void in this case. One small improvement that could be made to the code is that most variables are declared as unsigned 32-bit integers. For some variables, particularly in the secret key expansion function, do not need to be 32 bits. Initializing variables differently could save a few bits which could save some memory. Beyond that, the Ronald Rivest implementation is difficult to improve on for memory and execution time.

## Apple Open Source Implementation

Given the nature of RC5 as a simple algorithm, there are not many avenues to explore when trying to express RC5 differently. With the Apple implementation, the creators of the code decided to experiment with the versatility of the code. For example, the expand or set secret key function requires an input for key length and the number of rounds instead of utilizing parameterization. Theoretically, this still allows some level of versatility, but the implementation of the encryption and decryption function just functionally force the secret key expansion function’s inputs of rounds a fixed number. This is because number of rounds is hard coded into the decryption and encryption functions at 16 instead of dependent on a loop so a round number different from 16 would require edits to the decryption and encryption function every time. The rest of the secret key expansion is similar to the one outlined in Ronald Rivest’s paper with certain parts changed to account for the lack of parameterization. Beyond the glaring issue previously discussed, the two secret key expansion are quite similar in their implementation and perform as expected. With encryption and decryption though, the Apple Open-Source implementation is very different from the Ronald Rivest implementation. Whereas the Ronald Rivest implementation chose to have encryption and decryption done using a loop, the Apple Open-Source implementation chooses to hard-code 16 rounds of encryption only. The Apple implementation also has an input of an unsigned 8-bit number instead of 2 words. This forces the encryption and decryption to dedicate a few lines of code to assign A and B to the right values and another few lines of code to properly assign A and B back to the output of the function. The Apple implementation also neglects to index using brackets and a variable. Instead, the Apple implementation declares two different variables that point to specific values of the expanded secret key which increases the amount of memory required. With the Ronald Rivest implementation, only the indexing variable would have to be incremented but the Apple implementation requires two values to be adjusted every round. Compared with utilizing the bracket for expanded secret key indexing, only one value would need to be indexed. Having only one value would require a more complex operation, multiplication, to be performed twice every cycle to properly index so the Apple implementation is not always slower than the Ronald Rivest implementation.

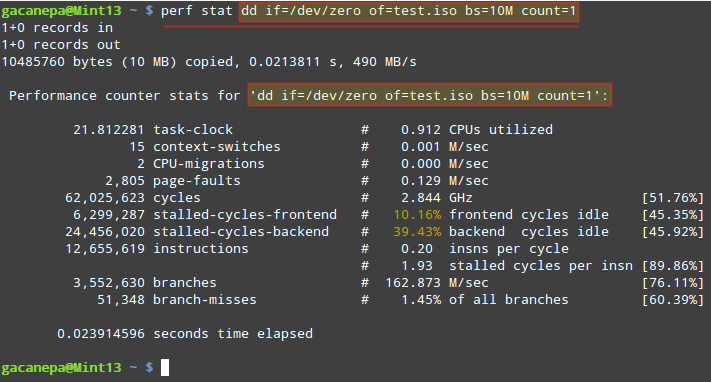
# Benchmarking

The benchmarking of the two different RC5 implementations was done on a laptop running Ubuntu 20.04.2 with a i5-6300 HQ @ 2.3 GHz and a Raspberry Pi 3 B+ running Raspbian GNU/Linux 10 (Buster) with a Cortex-A53 (ARMv8) 64-bit SoC @ 1.4GHz. Raspbian is an operating system designed to be used on Raspberry Pi software. Ubuntu is an operating system designed primarily for computers, smartphones, and network servers. Its requirements are more intense than Raspbian’s. With the Raspberry 3 B+, running the same Ubuntu version would be difficult. At the very least, the Raspberry Pi would have a reduced performance running the same version of Ubuntu that the laptop was running. The thought of using the same operating system for both systems did come up during this project, but that did not seem necessary. The purpose of this project is to benchmark the RC5 encryption algorithm on a Raspberry Pi 3. Benchmarking on a laptop was done simply to compare the performance of the Raspberry Pi to a much stronger machine. To sum things up, the version of Ubuntu used on the laptop is much more resource intensive than Raspbian which gives the Raspberry Pi a computation advantage, if any. Given that the purpose of this project is to benchmark the performance of RC5 on a Raspberry Pi, running a slightly more intense operating system on the laptop, just to have something to compare the Raspberry Pi’s performance to, is negligible.

## Perf

There are many different options to benchmark commands, such as oprofile or gprof, but perf was the most user friendly and easily installed so perf was chosen. Perf also is updated quite frequently, keeps up with specific kernels of Linux, and is included in a very convenient package with other linux tools. One key thing to note is that the laptop installation for perf is very simple and requires one command, sudo apt-get install linux-tools-generic. For the Raspberry Pi 3 B+, the kernel version is different and might not match the kernel version of perf. The workaround for this problem is to download a debian package for the matching kernel version of the Raspberry Pi. Running perf also becomes different. On the desktop, perf ‘EXAMPLE COMMAND’ will suffice but the debian installation only matches a specific kernel version so perf [version] ‘EXAMPLE COMMAND’ is necessary. Also note that perf might not work on Ubuntu because of security features included in perf. Changing the perf event paranoid value will fix this.

The specific command used to benchmark the RC5 algorithims was perf stat -r 100 ./rc5, perf stat -r 100 ./source, perf\_5.10 stat -r 100 ./rc5, and perf\_5.10 stat -r 100 ./source for the laptop and Raspberry Pi, respectively. The -r runs the command 100 times and then creates a table for the various statistics that it gathers. The following image is an example of the output that perf stat might yield:



A few key statistics that will be useful are task-clock, context switches, CPU migrations, cycles, instructions, and time elapsed. Context switches refer to the number of times that the state of a process or thread is stored so the process or thread can be restored and executed later. CPU-migrations refers to the case when a thread gets scheduled on a different CPU than it was scheduled before. In the case with the two different RC5 implementations, multi-threading is not utilized. As a result, CPU migrations and context switches should be 0. This is not too important a statistic in this case, but it is noteworthy that both of those values should be 0. In Patterson and Hennessy’s book, the terms CPU time and execution time are utilized. CPU time refers to time that only accounts for the time that the CPU is utilized for a task. Execution time refers to the total time that a computer takes to finish a task. Those terms are not present with the perf output, but task-clock and time elapsed are used instead. Task-clock is basically CPU time and execution time is the time elapsed. The task-clock value has no units stated in the example picture, but other perf commands verify that the task-clock units is in milliseconds. Cycles and instructions are fairly straightfoward and can be used to derive CPI. Note that the GHz value is obtained by dividing the number of cycles by the task-clock and might not be representative of the actual hardware’s clock capabilities.

## Raspberry Pi 3 B+ Benchmark

1. RC5

| Raspberry Pi 3 B+ | | |
| --- | --- | --- |
| Statistic | Value | Associated Statistic |
| Task-clock | 2.6 msec | 0.471 CPUs utilized (± 2.33%) |
| Context Switches | 0 | 0.000 K/sec |
| CPU-migrations | 0 | 0.000 K/sec |
| Page-faults | 45 | 0.0017 M/sec (± 0.18%) |
| Cycles | 900,671 | 0.346 GHz (± 1.66%) |
| Instructions | 363,634 | 0.4 insn per cycle (± 0.00%) |
| Branches | 38,750 | 14.899 M/sec (± 0.00%) |
| Branch-misses | 7,521 | 19.41% of all branches (± 0.29%) |
| Time Elapsed | 0.005521 | ±0.000203 (±3.67%) |

(1)

1. Apple

| Raspberry Pi 3 B+ | | |
| --- | --- | --- |
| Statistic | Value | Associated Statistic |
| Task-clock | 2.62 msec | 0.486 CPUs utilized (± 2.54%) |
| Context Switches | 0 | 0.000 K/sec |
| CPU-migrations | 0 | 0.000 K/sec |
| Page-faults | 45 | 0.0017 M/sec (± 0.19%) |
| Cycles | 912,460 | 0.349 GHz (± 1.68%) |
| Instructions | 378,781 | 0.42 insn per cycle (± 0.00%) |
| Branches | 42,161 | 16.114 M/sec (± 0.00%) |
| Branch-misses | 7,494 | 17.77% of all branches (± 0.31%) |
| Time Elapsed | 0.005388 | ±0.000189 (±3.51%) |

(2)

## Laptop Benchmark

1. RC5

| Laptop | | |
| --- | --- | --- |
| Statistic | Value | Associated Statistic |
| Task-clock | 0.4 msec | 0.656 CPUs utilized (± 0.87%) |
| Context Switches | 1 | 0.0002 M/sec (± 37.17%) |
| CPU-migrations | 0 | 0.000 K/sec |
| Page-faults | 55 | 0.138 M/sec (± 0.14%) |
| Cycles | 1,012,928 | 2.564 GHz (± 0.49%) |
| Instructions | 963,889 | 0.95 insn per cycle (± 0.27%) |
| Branches | 193,606 | 489.977 M/sec (± 0.31%) |
| Branch-misses | 7,259 | 3.75% of all branches (± 0.62%) |
| Time Elapsed | 0.0006153 | ±0.0000300 (± 4.88%) |

(2)

1. Apple

| Laptop | | |
| --- | --- | --- |
| Statistic | Value | Associated Statistic |
| Task-clock | 0.38 msec | 0.614 CPUs utilized (± 0.55%) |
| Context Switches | 1 | 0.002 K/sec (± 31.89%) |
| CPU-migrations | 0 | 0.000 K/sec |
| Page-faults | 55 | 0.145 M/sec (± 0.15%) |
| Cycles | 1,028,259 | 2.722 GHz (± 0.48%) |
| Instructions | 960,094 | 0.93 insn per cycle (± 0.19%) |
| Branches | 192,816 | 510.473 M/sec (± 0.21%) |
| Branch-misses | 7,336 | 3.8% of all branches (± 0.46%) |
| Time Elapsed | 0.0006028 | ±0.0000156 (± 2.59%) |

(2)

## Power

The power used by the Raspberry Pi 3 B+ was gathered using an USB ammeter. There are two external devices connected to the Raspberry Pi 3 B+ during testing, a keyboard and mouse. The keyboard has a voltage of 5.1538 V and a current of about 0.2279 A for a power consumption of 1.1746 W. The mouse had a voltage of 5.2000 V and a current of about 0.0249 A for a power consumption of 0.1295 W. With both external devices connected, the Raspberry Pi 3 B+ has a voltage of 5.2100 V and a current of about 0.6638 A which is 3.4584 W. Without the external devices, the Raspberry Pi 3 B+ has a voltage of 5.2500 V and a current of about 0.4700 A, which is 2.64 W. With this information, the external devices result in a 31% increase in the power consumed by the Raspberry Pi 3 B+, compared to when the Raspberry Pi 3 B+ has no external devices connected.

Before execution of the Apple implementation, the Raspberry Pi 3 B+ had a voltage of 5.21 V and a current of about 0.6638 A for a power of 3.4584 W. During execution, the Raspberry Pi 3 B+ has the same voltage, 5.21 V, but the current draw increased to 0.7300 A for a power of 3.8033 W. Therefore, the Apple implementation on the Raspberry Pi 3 B+ required 0.3449 W to perform encryption, decryption, secret key formation, and verification of results.

Before execution of the Ronald Rivest implementation, the Raspberry Pi 3 B+ had a voltage of 5.21 V and a current of about 0.6624 A for a power of 3.4511 W. During execution, the Raspberry Pi 3 B+ had the same voltage, 5.21 V, but the current increased to 0.7700 A for a power of 4.0117 W. Therefore, the Ronald Rivest implementation required 0.5606 W to perform encryption, decryption, secret key formation, and verification of results.

The Apple implementation took 0.3449 W to perform encryption, decryption, secret key formation, and verification of results which is 0.2157 W more than the Ronald Rivest implementation. In other words, the Ronald Rivest implementation required 62.54% more power than the Apple implementation or the Apple implementation required 38.48% less power than the Ronald Rivest implementation to complete the same task.

## Remarks

Over 100 executions on the Raspberry Pi 3 B+, the Ronald Rivest implementation had a shorter CPU execution time, but the Apple implementation had a shorter wall clock time. The Ronald Rivest implementation had a greater CPI but had overall less cycles and instructions than the Apple implementation. On the laptop, the Apple implementation had a lesser CPU execution time, and the Ronald Rivest implementation had a shorter wall clock time. Comparing both the laptop and the Raspberry Pi 3 B+, the laptop had a much faster execution and wall clock time which was to be expected. The laptop had a CPU @ 2.3 GHz which is about 1.6429 times faster than the Raspberry Pi’s CPU @1.4 GHz. The laptop’s execution time was about 6.5 times faster than the Raspberry Pi while the wall clock time was about 8.75 times faster. The laptop running the two different implementations also had some background applications running, which could have influenced the benchmarking. Given that the task-clock, context switches, and CPU-migrations are almost as expected and that the wall clock time was not too far off from the CPU execution time, the extraneous applications must not have had too big of an impact on the benchmarking. The Raspberry Pi B+ did not have any background applications running, though the peripherals might have decreased the amount of power available to the Raspberry Pi and decreased its performance.

Regarding power, the Apple implementation was much more power efficient than the Ronald Rivest implementation. The power draw was from the external devices was significant, but it is uncertain whether the power draw was high enough to impact the performance of the Raspberry Pi. The Pi’s performance is noticeably slow with no applications running with the same peripherals, keyboard and mouse, attached. Nonetheless, both the Apple implementation and the Ronald Rivest implementation had the same external attached during the benchmarking so both implementations would have experienced the same effects. One thing to consider is that perhaps the benchmarked data is only accurate for a Raspberry Pi 3 B+ under power constrained conditions, but that is uncertain.

To conclude, the Ronald Rivest implementation and the Apple implementation, over 100 runs, perform roughly the same, compared to each other, on the laptop and the Raspberry Pi 3 B+. With power, the Ronald Rivest implementation requires up to 62.54% more power than the Apple implementation.

##### References

1. Rivest, R., 1997. *The RC5 Encryption Algorithm*. [online] Cambridge: MIT Laboratory for Computer Science. Available at: <https://people.csail.mit.edu/rivest/Rivest-rc5rev.pdf> [Accessed 18 March 2021].
2. <https://opensource.apple.com/source/xnu/xnu-201/bsd/crypto/rc5/>
3. Hua Li, Jianzhou Li and Jing Yang, "An efficient and reconfigurable architecture for RC5," Canadian Conference on Electrical and Computer Engineering, 2005., 2005, pp. 1648-1651, doi: 10.1109/CCECE.2005.1557299.
4. T. Nie, Y. Li and C. Song, "Performance Evaluation for CAST and RC5 Encryption Algorithms," 2010 International Conference on Computing, Control and Industrial Engineering, 2010, pp. 106-109, doi: 10.1109/CCIE.2010.34.
5. Alex Biryukov, and Eyal Kushi, “Improved Cryptanalysis of RC5,” EUROCRYPT ’98., 1998, pp. 85-89, doi: 10.1007/BFb0054119

**IEEE conference templates contain guidance text for composing and formatting conference papers. Please ensure that all template text is removed from your conference paper prior to submission to the conference. Failure to remove template text from your paper may result in your paper not being published.**