TEA Performance (May 2021)

Sydney Cady, Ryan Schwartz, Steven Yu, Mohamed El-Hadedy  
  
California State Polytechnic University, Pomona, College of Engineering, ECE Department  
  
rmschwartz@cpp.edu, slcady@cpp.edu, stevenyu@cpp.edu, mealy@cpp.edu

[[1]](#footnote-1)

*Abstract*—The main objective for this project is to better understand the performance of TEA (Tiny Encryption Algorithm) and how it can be applied to portable IoT solutions, such as the popular Raspberry Pi device. Within this project we will be analyzing different performance metrics like, CPI, MIPS, execution time, CPU rate, cycles per second, and power metrics of the execution of the TEA algorithm on a Raspberry Pi. Within this project, we demonstrate NUMBERS of results, power stuff, how much can power the portable device, etc.

Keywords – Tiny Encryption, TEA, Raspberry Pi,

# INTRODUCTION

The authors of this paper collectively enrolled and studied in the *Computer Architecture* lecture offered at Cal Poly Pomona. Demonstrated throughout this paper, the inner workings of the exact architecture that goes into computers and their operating system are better understood and the ways in which things can be more efficient or in the ways that efficiency is truly measured specifically using the TEA algorithm on the Raspberry Pi. As an example of TEA in public use today in the world, \*slow encryption solution in public? any use of tea in public now?\*. Our group was assigned to better understand the Tiny Encryption Algorithm, or TEA, in which we will do so throughout this report. The performance of the TEA is analyzed by using different languages on a Raspberry Pi 3B+ and analyzing the MIPS, M flip-flops, execution time, CPU rate, cycles per second, and CPI, as well as additional power data collected while the Raspberry Pi is under light and heavy load with the TEA. Through doing this, we will better understand TEA and will demonstrate its usefulness on a portable IoT device such as the Raspberry Pi 3B+.

# Related Work

Compare this paper to other papers.

# Background

Most of our understanding comes from understanding the few lectures within this class that helps us understand the metrics within computer performance. By using this data we are able to understand how our programs function and the measurements we can take. So in this project we use a Raspberry Pi 3B+ and use Python and C++. By doing this we will hopefully be able to measure the metrics of each language and compare it to one another. This will be done through timers within our system and calculations with the data we find of each program. Some calculations would be for the MIPS for example. We will also be taking measurement with other tools like a power meter in order to establish how many encryption encipher-decipher pairs can be executed within specific boundaries or limitations.

# Project Details

This section will go over the understanding of TEA through explanation and graphics. Then will go over our execution of the TEA algorithm in Python and C++. Once we have established the basics we will demonstrate the different metrics we were able to capture.

## TEA Background

So, what is TEA? TEA is the Tiny Encryption Algorithm. TEA is a cryptographic algorithm in which its goals is be efficient in memory footprint and speed. TEA is mainly noted for its simplicity as it can be executed with only a few lines of code. Which is impressive because it has a Feistel structure which means that it could have about 64 rounds, or known as cycles, in encrypting the key. It is incredibly simple because each cycle will be mixed in the exact same way as is set by the key schedule. There are ways to prevent simple attacks, like implementing a magic constant since simpler attacks would be based on the symmetry of TEA’s algorithm.

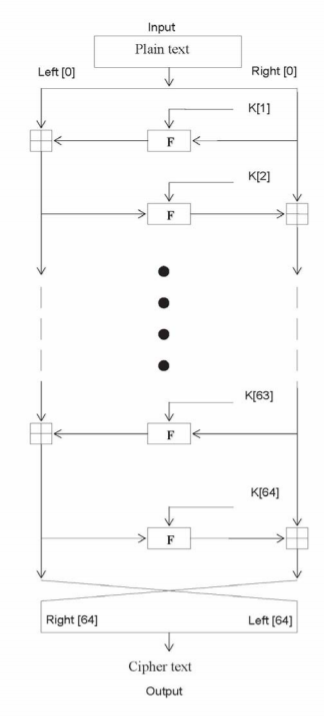


Fig 1. Here is an abstract structure of the TEA routine. As you see it go through about 64 cycles before reaching the end.

To take a deeper understanding into the TEA algorithm, we can further look into an i-th iteration of the cycle that a key would go through.

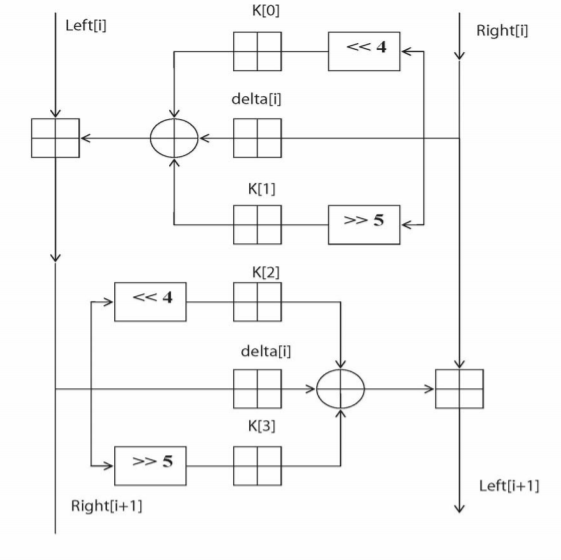


Fig. 2. This is an abstract view of the i-th iteration of the TEA cycle.

To understand what these last two figures represent we must first understand what each notation means. Each round (i) will have a left and right input, i-1, which is derived from the previous round or cycle. The sub keys are K[i] which differs from K and each other, but K[i] is also derived from the overall 128 bit key input K (2 64-bit keys). The constant delta is equivalent to which comes from the golden number ration. This number helps ensure that sub keys are unique.

## TEA Code Execution

In this section we will supply the code used to take measurements for our project.

The code used in Python is displayed below:

|  |
| --- |
| import sys  from ctypes import \*  import time  import cProfile  #from memory\_profiler import profile  def encipher(v, k):  t1\_encipher = time.perf\_counter()  y = c\_uint32(v[0])  z = c\_uint32(v[1])  sum = c\_uint32(0)  delta = 0x9e3779b9  n = 32  w = [0,0]  while(n>0):  sum.value += delta  y.value += ( z.value << 4 ) + k[0] ^ z.value + sum.value ^ ( z.value >> 5 ) + k[1]  z.value += ( y.value << 4 ) + k[2] ^ y.value + sum.value ^ ( y.value >> 5 ) + k[3]  n -= 1  w[0] = y.value  w[1] = z.value  t2\_encipher = time.perf\_counter()  #print(f"Enchipering took {t2\_encipher - t1\_encipher:0.4f} seconds.")  return w  def decipher(v, k):  t1\_decipher = time.perf\_counter()  y = c\_uint32(v[0])  z = c\_uint32(v[1])  sum = c\_uint32(0xc6ef3720)  delta = 0x9e3779b9  n = 32  w = [0,0]  while(n>0):  z.value -= ( y.value << 4 ) + k[2] ^ y.value + sum.value ^ ( y.value >> 5 ) + k[3]  y.value -= ( z.value << 4 ) + k[0] ^ z.value + sum.value ^ ( z.value >> 5 ) + k[1]  sum.value -= delta  n -= 1  w[0] = y.value  w[1] = z.value  t2\_decipher = time.perf\_counter()  #print(f"Deciphering took {t2\_decipher - t1\_decipher:0.4f} seconds.")  return w  def encipher\_decipher(v, key):  enc = encipher(v, key)  #print(enc)  #print(decipher(enc, key))  decipher(enc, key)  if \_\_name\_\_ == "\_\_main\_\_":  t1 = time.process\_time\_ns()  key = [1,2,3,4]  v = [1385482522,639876499]  for num in range(0, 101):  encipher\_decipher(v, key)  #enc = encipher(v,key)  #print (enc)  #print (decipher(enc,key))  t2 = time.process\_time\_ns()  print("50 TEA Encipher-Deciphers executed in " + str(t2-t1) + " ns / " + str((t2-t1)/1000000000) + " s.")    #cProfile.run('encipher\_decipher()') |

To get a certain metric on the Python TEA code, we had to convert the Python into assembly. This was done through using a website called godbolt.org. Through this website we were able to *roughly* determine that the Python assembly code would have 348 instructions.

The code used in the C++ is displayed below:

|  |
| --- |
| #include <stdio.h>  #include <chrono>  #include <iostream>  void encipher(unsigned long \*const v,unsigned long \*const w, const unsigned long \*const k)  {  unsigned long y=v[0],z=v[1],sum=0,delta=0x9E3779B9, a=k[0],b=k[1],c=k[2],d=k[3],n=32;  while(n-->0)  {  sum += delta;  y += (z<<4)+a ^ z+sum ^ (z>>5)+b;  z += (y<<4)+c ^ y+sum ^ (y>>5)+d;  }  w[0]=y; w[1]=z;  }  void decipher(unsigned long \*const v,unsigned long \*const w,  const unsigned long \*const k)  {  unsigned long y=v[0],z=v[1],sum=0xC6EF3720,  delta=0x9E3779B9,a=k[0],b=k[1],c=k[2],  d=k[3],n=32;  /\* sum = delta<<5, in general sum = delta \* n \*/  while(n-->0)  {  z -= (y<<4)+c ^ y+sum ^ (y>>5)+d;  y -= (z<<4)+a ^ z+sum ^ (z>>5)+b;  sum -= delta;  }  w[0]=y; w[1]=z;  }  int main()  {  auto start = std::chrono::high\_resolution\_clock::now();  unsigned long v[] = {0xe15034c8, 0x260fd6d5};  unsigned long key[] = {0xbe168aa1, 0x16c498a3, 0x5e87b018, 0x56de7805};  unsigned long res[2];  for (int i = 0; i < 50; i++)  {  encipher(v, res, key);  //printf("Enciphered: %X %X\n", res[0], res[1]);  std::cout << "Loop #" << i + 1 << std::endl;  decipher(v, res, key);  //printf("Deciphered: %X %X\n", res[0], res[1]);  }  auto stop = std::chrono::high\_resolution\_clock::now();  auto duration = std::chrono::duration\_cast<std::chrono::nanoseconds>(stop-start);  std::cout << "50 TEA Encryption-Decryptions took " << duration.count() << " nanoseconds\n\n";  return 0;  } |

To get certain metric on the TEA code, we had to convert the C++ into assembly. This was done through utilizing the GNU GCC compiler and running the command “g++ -S <filename>” in the command line, which would output the assembly code. By doing so we found that the C++ TEA code had about 687 instructions. Note: gathering assembly code from C++ is much easier and 100% accurate as C++ is compiled directly into machine language, whereas Python assembly code was supplied by a rough estimate of what the instructions would generally look like, as Python is never translated into machine code with a compiler (it is instead an interpreted language).

# TEA Metrics Tables

We have been able to establish the basics form this point. Now this section will go into the different metrics we found. Here will be a table of all the metrics we have collected. To begin, we’ll provide some general metrics that are applied to the Raspberry Pi.

|  |  |
| --- | --- |
| Raspberry Pi Data | Value |
| Clock rate (max) | 1.2MHz |
| Clock cycle time | 8.333 E-7 |

Table 1. Raspberry Pi metrics

Here is a table of the Python data collected on the Raspberry Pi:

|  |  |
| --- | --- |
| Python Data | Value |
| Assembly instruction count | 387 |
| CPU/Execution time | .0016983771 seconds |
| CPU clock cycles | 2038.13 |
| CPI | 5.26629 |
| MIPS | 0.227865 |

Table 2. Python metrics collected with TEA on Raspberry Pi

Here we will have a table of the C++ Data:

|  |  |
| --- | --- |
| C++ Data | Value |
| Assembly instruction count | 687 |
| CPU/Execution time | .000145586 seconds |
| CPU clock cycles | 174.71 |
| CPI | 0.254299 |
| MIPS | 4.71886 |

Table 3. C++ metrics collected with TEA on Raspberry Pi

These tables are filled with performance metrics that this report will further go into in the next section where we will show the math behind some of the performance metrics.

# TEA Data Collection

In this section we will be going over: Clock cycles, CPI, and MIPS. We will be demonstrating the equations we used in order to understand these performance metrics in TEA.

## CPU Clock Cycles

First off, lets determine the CPU clock cycles based upon whether or not it is Python or C++.

The basic equation we can use is below:

Thus, for Python we can use this equation, by substituting the CPU execution time and clock cycle time from the python and Raspberry Pi tables. Which will give us:

Then with C++, we use the same equation to determine the CPU clock cycles.

## CPI

Then, we can use the previous data to determine the CPI for Python and C++. This is done through the CPI equation as shown below:

Thus, by using this equation we can determine the CPI for Python and C++. First let’s determine the CPI for Python:

Then for C++ we can determine the CPI:

## MIPS

Using another equation, we can determine the MIPS, millions of instructions per second. This is done through this equation below:

Through this equation we can find the MIPS for the Python code:

0.227865

Then by applying the same logic to C++, we get that the MIPS is:

4.71886

## Power Analysis

Instead of executing a fixed number of encipher-decipher pairs, each program was run for a total of 12 minutes. 12 minutes is an arbitrary length of time but is a long enough time to see the resulting curves of the graphs and extrapolate data from them. As shown in the pictures to follow, each “block” is 50 encipher-decipher pairs. Multiplying the block number by 50 is the total amount of enciphers and deciphers performed by the Raspberry Pi. Power consumption was tested using the least efficient method, Python, with 1 and 3 TEA programs executing at a time. Lastly, voltage, current, and power consumption data was gathered every 2 minutes.

For only a single TEA program executing, there were a total of 8985 blocks over 12 minutes. This equates to a total of 449,250 total enciphers and deciphers.

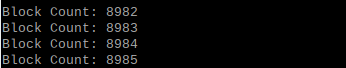


Fig. 3. Block Count From One TEA Process(12 minutes)

|  |  |  |  |
| --- | --- | --- | --- |
| Time (min.) | Voltage (V) | Current (A) | Power (Wh) |
| 2 | 4.75 | .34 | .05 |
| 4 | 4.83 | .34 | .11 |
| 6 | 4.84 | .4 | .17 |
| 8 | 4.84 | .41 | .23 |
| 10 | 4.85 | .41 | .29 |
| 12 | 4.83 | .39 | .35 |

Table 4. Voltage, current, and power data was captured and recorded in the table below with intervals of 2 minutes each. This is in regards with Python with singular encryption, no parallelism.

As seen from the table above, voltage and current remained constant throughout the entire 12 minutes; voltage with a 4.82V average, current with a .38A average. Power linearly rose over the 12-minute trial depicted in the Power Consumption vs. Time graph below.

Fig. 5. Voltage vs. Time graph. This is in regards with Python with singular encryption, no parallelism.

Fig. 6. Current vs. Time graph. This is in regards with Python with singular encryption, no parallelism.

Fig. 7. Power Consumption vs. Time graph. This is in regards with Python with singular encryption, no parallelism.

There was also the analysis of the relationship between the total amount of encipher-decipher pairs and the amount of power consumed. Surprisingly, this relationship is strictly linear as can be seen in the graph below.

|  |  |  |
| --- | --- | --- |
| Time (min.) | Encipher-Decipher Pairs | Power (Wh) |
| 2 | 74875 | .05 |
| 4 | 149750 | .11 |
| 6 | 224625 | .17 |
| 8 | 299500 | .23 |
| 10 | 374375 | .29 |
| 12 | 449250 | .35 |

Table 5. Power analyzed through the encipher and decipher pairs. This is in regards with Python with singular encryption, no parallelism.

Fig. 8. Encipher-Decipher pairs vs power consumption. A graphical view of table 5. This is in regards with Python with singular encryption, no parallelism.

Fig. 9. A stacked chart comparing voltage, amperes, and watthours. This is in regards to Python with singular encryption, no parallelism.

Next, three TEA processes were run simultaneously in order to determine if the Raspberry Pi struggles with parallel execution and to watch for any new trends in voltage, current, or power metrics. Again, voltage, current, and power data was collected every 2 minutes.

For three parallel TEA programs executing, there were a total of 7932 + 7957 + 7938 = 23827 blocks over 12 minutes. This equates to a total of 1,191,350 total enciphers and deciphers.

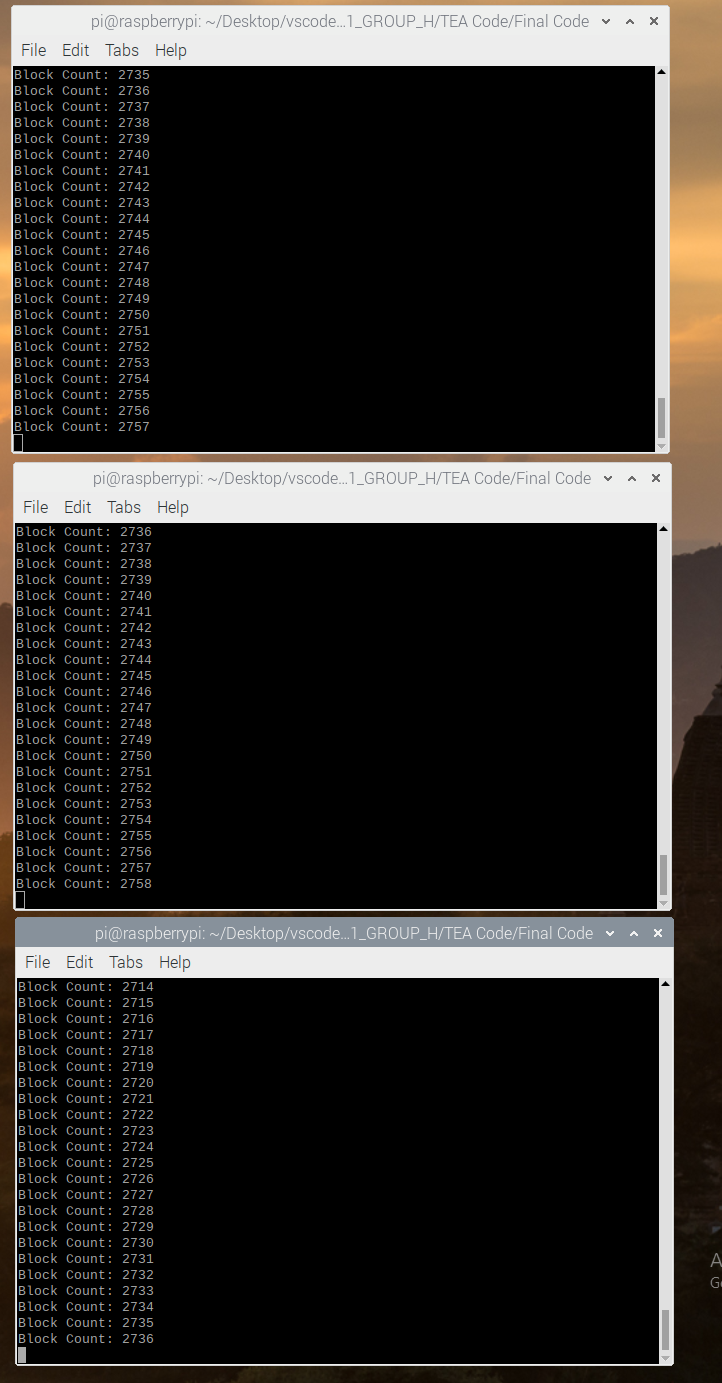








Fig. 10.Three Block Counts From Parallel Execution (12 minutes)

Again, voltage, current, and power data was captured and recorded in the table below with intervals of 2 minutes each.

|  |  |  |  |
| --- | --- | --- | --- |
| Time (min.) | Voltage (V) | Current (A) | Power (Wh) |
| 2 | 4.78 | .53 | .08 |
| 4 | 4.7 | .48 | .16 |
| 6 | 4.69 | .54 | .24 |
| 8 | 4.72 | .54 | .33 |
| 10 | 4.62 | .54 | .41 |
| 12 | 4.74 | .58 | .49 |

Table 6. Voltage, current, and power data was captured and recorded in the table below with intervals of 2 minutes each. This is in regards with Python with parallelism execution.

Just like the individual TEA process, the three parallel processes required a steady voltage and current from the device; voltage with a 4.71V average, current with a .54A average Again, power linearly rose over the 12-minute trial depicted in the graphs below.

Fig. 11. Voltage Vs. Time graph. This is in regards with Python with parallelism execution.

Fig. 12. Current vs. Time graph. This is in regards with Python with parallelism execution.

Fig. 13. Power Consumption vs. Time graph. This is in regards with Python with parallelism execution.

Additionally, there was also the analysis of the relationship between the total amount of encipher-deciphers and the amount of power consumed. Similarly with the single program execution, relationship is strictly linear as can be seen in the graph below.

|  |  |  |
| --- | --- | --- |
| Time (min.) | Encipher-Decipher Pairs | Power (Wh) |
| 2 | 198558.3 | .08 |
| 4 | 397116.7 | .16 |
| 6 | 595565 | .24 |
| 8 | 794123.3 | .33 |
| 10 | 992681.7 | .41 |
| 12 | 1191350 | .49 |

Table 7. Power analyzed through the encipher and decipher pairs. This is in regards with Python with singular encryption, no parallelism.

Fig. 14. Encipher-Decipher pairs vs power consumption. A graphical view of table 5. This is in regards with Python with parallelism execution.

Fig. 15. A stacked chart comparing voltage, amperes, and watthours. This is in regards with Python with parallelism execution.

## Power Analysis Photo Captures

Below shown is all the photos taken with the Power Meter to show the data we collected.







Fig. 16. Photos of single TEA process running.







Fig. 17. Photos of three TEA processes running in parallelism.

## Applicable Portable Power-bank Scenario

A power bank was used to test the power output and power manageability of the TEA algorithm on the Raspberry Pi 3B+. This power bank was a TYLT model WA-15TPP5200. It is capable of outputting the necessary 5 volts, as well as 1 ampere of maximum current. As described above, the TEA requires 4.82 volts and 0.38 amperes on average with one single algorithm process running. Because of this 0.38 A average, the expected time to drain this portable power bank using the TEA on a Raspberry Pi would be a long   
5200mAh / 380 mA = 13.7 hours. Instead of watching the power test for such a long period, the three TEA processes were running at 0.54A in order to fasten the test by 29.6%. Extrapolating this speed-up from the 13.7 hours expected from the single process, 13.7 – (13.7 \* .296) = 9.6 hours. Again, given the linearly relationship between power consumed and the amount of encipher-deciphers performed, it is expected from the team that any linear division of the time of 9.6 hours will accurately describe the power consumed on a portable power bank. To illustrate, if 9.6 hours is the expected run time on a fully charged 5200mAh power bank outputting 540mA to the Raspberry Pi, then at 4.8 hours into the test, we should see the power bank at half charge. This is the final decision of the team because of the more reasonable requirement of the sitting at a desk and making sure the power bank continues to deliver power to the Raspberry Pi, as well as making sure the Raspberry Pi remains in normal operating mode of the three processes.

As a result, after 4 hours and 48 minutes of using a fully charged 5200mAh power bank to power the Raspberry Pi 3B+ running three concurrent TEA processes, the power bank correctly showed it was at two out of four “bars” left on the battery capacity as shown below (the two leftmost bars are bright in the picture, although it may be hard to see).

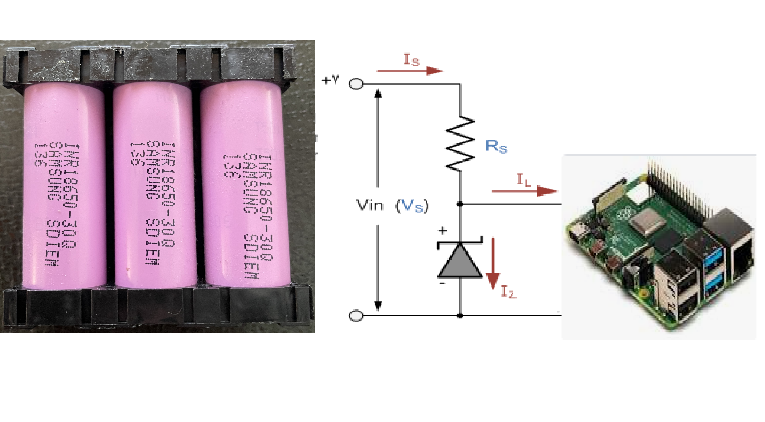


This result agrees with the proposed half-capacity theory of 4.8-hour power bank usage. That being said, there may be error in this analysis due to the power bank on-hand for the team may or may not having an accurate percentage display of the capacity left in the power bank, just bars. Two out of four bars *could* indicate that the capacity remining in the power bank could be anywhere between 65% to 40%, for example. Nevertheless, our theory is correct with the data shown by the portable power bank.

Instead of an expensive power bank, it is theorized by the team that an individual could power the Raspberry Pi 3B+ using multiple of the popular 18650 battery. 18650 batteries are the standard in today’s market when it comes to rechargeable devices, so it may be more likely for an average household to contain 18650 batteries instead of power banks that specifically output 5V at 540mA or more. That being said, a thought experiment was conducted to test whether 18650 batteries could deliver portable power to the Raspberry Pi. A power versus time test was not conducted because the portability and capacity testing were conducted in the previous paragraphs of this section.

The specific 18650 batteries used in this thought experiment were Samsung 30Q 3000mAh 18650s. To start the trial, 3 18650 batteries would be collected and connected in series. The reason for this is as follows; each battery has a cutoff voltage of 2.5V, so two batteries in series could theoretically give the 5V needed to power the Raspberry Pi, however, the team would not want to damage these batteries by draining them to their absolute minimum, so three batteries were connected in series in order to preserve the life of the batteries and allowing the batteries to only discharge to a minimum of whatever their battery management system (BMS) thinks is appropriate. Also, each battery has a max continuous discharge of 15A, so there is no bottleneck in terms of current discharge.

After the power supply of the 18650 batteries is figured out, the required maximum current and the resistor to the load will need to be calculated. First, the maximum current delivered to the load will be the average current of the three TEA processes during the 12-minute trial, 0.54 amperes. Next, the series resistor can easily be calculated using Ohm’s law, . As seen in the previous equation, a 14.1 ohm resistor can be used to deliver 540 mA of current to the 5V output when the 18650 batteries are fully charged in series for a 12.6V power supply. If a situation occurs where the cells begin to discharge, e.g. if the cells are not connected to a constant current constant voltage (CCCV) power supply or the BMS attached to the cells allow the cells to drop to a safe 3V, then, in that example, each cell would have a minimum of 3V, so three cells in series would give 9V power supply. Again, the equation will show . This will result in a 7.5 ohm resistor delivering 540 mA of current to a 5V output. To meet somewhere in the middle, a 10 ohm resistor could be used to achieve a reasonable current close to 540mA without much drop off of power. A theorized circuit will look something like the picture shown below.



Assume the above picture to have three 30Q 18650 cells attached in series as the power supply, along with a Zener diode acting as a voltage regulator, and Rs acting as a simple current regulator, with all output directed towards the Raspberry Pi. With this setup, a user can expect to achieve 5.6 hours of a power supply from these 3000mAh capacity batteries. Although less than the other power bank solution, 18650 batteries can be added in parallel to increase capacity! If the application requires it, the batteries can be added in parallel for as big as the user wants. Each cell in parallel *multiplies* the capacity of the system by that number. For example, a 3S3P system, with 3 cells in series as before, but 3 cells in parallel as well, would triple with capacity allowing the system to have a total of 9000mAh storage capacity and supplying a power supply time of 16.7 hours. An example of an overkill 3S5P power supply with 15000mAh of capacity is shown below. This solution would give 28 hours of power before requiring to be quickly recharged.



## Amdahl’s Law: TEA vs. XTEA

XTEA or the Extended TEA is an improved version of TEA that aims to solve the vulnerabilities of original algorithm. XTEA is designed to avoid the “equivalent keys” weakness of the original TEA in which each key is equivalent to 3 others, thus reducing the effective key size from 128 bits to 126 bits. XTEA also defends against “related key attacks” that the original TEA is vulnerable to.

In this benchmark the overall speedup from the original TEA to the improved XTEA will be calculated using Amdahl’s law. Both the TEA and the XTEA code in this test are written in C++. Amdahl’s law is as detailed:

The execution time of the XTEA program is calculated by running 50 encryption-decryption pairs and timing the entire duration. The duration is then divided by 50 runs to find the average per-pair execution time.

We can take the values found from XTEA, the execution time: 0.0000166539 seconds, and the assembly line count is measured to 256 lines. Thus we can use the Amdahl’s Law Overall Speedup equation to see the overall speedup.

Now we can implement the data. The fraction would be new/old = XTEA assembly line count / TEA assembly line count. Then, to find S, the speedup of enhanced fraction, we can use the value from before where it explains to us how much faster the program runs.

Thus, the overall speedup is 1.03036 from these calculations. We can guarantee and show that XTEA performs much better than TEA as it essentially performs almost 10 times faster and has an incredible speedup compared to the original TEA.

# Conclusion

Through this experiment we were able to come up with a couple different conclusions when working with TEA and Python or C++. First off, when analyzing the Python assembly count, we were able to determine that there aren't many accurate resources that will convert Python into assembly. As within Raspberry Pi, Python gets converted into C then assembly to be understood, so we attempted to break down our python code into assembly. The one resource we used stated that the Python code equated to about 400 lines of assembly even though it may have been more or less. We can infer this because the generic C++ code is almost 800 lines of code and runs ten times quicker than the Python on the Raspberry Pi. Thus, even though we used this amount of instructions as a metric for our data, we are unsure as to how accurate it is because the resources are inaccurate.

As stated above, we were able to conclude that the C++ code runs ten times faster than the Python code. The C++ is able to execute its code at a fast .000145586 seconds whereas the Python is a very slow .0016983771 seconds. to be more exact, the C++ code runs 11.7 times faster. This helps understand the other metrics found within our data. For example, we can understand that C++ is more efficient at cycling the instructions as shown through the CPI data found and the execution time. Whereas compared to the Python time, it takes a much longer time to run and has many more cycles to run per instruction (as found in the CPI), as well as the fact that we can analyze the MIPS (million instruction per second) with the understanding of the execution time. For example, with the faster run time, we may be able to infer that the MIPS is pretty high. For C++, we calculated the MIPS and received a fairly high MIPS. Whereas when the system ran slower we received a smaller MIPS, like with Python. A couple things get considered here though, like the execution time and the amount of instructions, so C++ is most likely a more accurate understanding of MIPS whereas Python highly depended on an unreliable resource to determine the assembly instruction count.

Through using a power meter attached to a Raspberry and TEA, we were able to get a better understanding of how TEA performs with power consumption. . Interestingly, with three processes running, the average voltage is *lower* than a single process running by about 2.5%, but the current is *higher* than a single process running by 30%.

After conducting the applicable scenario of powering the Raspberry Pi with a power bank and the theoretical thought experiment of powering the Raspberry Pi with popular 18650 batteries, it is found that each scenario would work well. Powering with an appropriate 5V output power supply, it was found that the Raspberry Pi running three parallel TEA processes at once could run for a total of 9.6 hours. Although our findings proved to be correct during this experiment, there could always be error given that the power bank did not display clear and accurate capacity measurements and energy could also be lost in the process due to natural inefficiencies. On top of that, the thought experiment of three 18650 battery cells in parallel seems to be another appropriate method of powering the Raspberry Pi. Three cells in series, giving a max voltage of 12.6V and minimum voltage of, for example, 9V, can easily be constructed using the diagram shown of a basic Zener diode in series with a calculated 7.5ohm to 14.1ohm resistor. Although three cells in series only gives a capacity of 3000mAh allowing the Raspberry Pi to run for a total of 5.6 hours (given an average 540mA to power the three simultaneous TEA processes), it was demonstrated that this solution is much more easily scalable. More 18650 battery cells can be assorted in parallel to keep the output voltage the same but increase the capacity of the power supply. A 3S5P power supply was shown that would keep the three cells in series to maintain the 9V-12V but increase the capacity by a factor of 5! The new capacity in this case would be 15000mAh that could power the Raspberry Pi for a total of 28 hours if needed.

The clock rate of the Raspberry Pi 3B+ is 1.2MHz at max clock speed which means the cycle time is 1/1.2MHz = 8.33 x 10^-7. For C++ , the execution time for 50 encipher-decipher pairs took a total of 7279302 nanoseconds, which equates to .000145586 seconds over a total of 687 assembly instructions. For Python, the execution time for 50 encipher-decipher pairs took a total of 84918855 nanoseconds, which equates to .0016983771 seconds over a total of 348 instructions. Dividing the Python execution time by the C++ execution time, . 0016983771 / .000145586 = 11.67, we can see that the C++ program is faster than Python by a factor of 11.67 on 32-bit Raspberry Pi OS.

References

1. Python TEA implementation, stack overflow, <https://stackoverflow.com/questions/2588364/python-tea-implementation> (accessed March 17, 2021)
2. David Wheeler and Roger Needham, “The Tiny Encryption Algorithm (TEA)”, Cambridge Computer Laboratory
3. Python to Assembly, Compiler Explorer, <https://godbolt.org/> (accessed March 20, 2021)

**Sydney Cady**  was born in Mission Viejo, California. She is currently studying to receive her B.S. in Computer Engineering at California State Polytechnic University, Pomona, which is expected to be earned in December of 2021. She has been on his university’s President’s Honors List three years in a row, from his first year to her third year. She has been a part of the Kellogg’s Honor College for 2 years where excellence and community engagement is valued.

Her research interests include software development, machine learning, and object-oriented concepts, where she has been able to apply to the various project done on campus. She has participated in a virtual internship at AT&T.

**Ryan Schwartz** was born in Oxnard, California. He is currently studying computer engineering at California State Polytechnic University, Pomona and is graduating in Fall 2021. He has experience with PIC18F microcontrollers supplied by his curriculum and has worked with the popular Raspberry Pi and Arduino microcontrollers through personal projects.

He has had more professional experience through few of his previous internships where he has learned for about system architectural development for mission control, mission data, and mission information systems, as well as more software development experience through machine learning techniques such as natural language processing and Windows CRM online servers for marketing and production.

**Steven Yu** was born in Anaheim, California. He is currently studying in the Department of Electric and Computer Engineering at California State Polytechnic University, Pomona. He is expected to graduate with a B.S. in Computer Engineering. Some of his interests include developing robotics and electronic devices.

1. [↑](#footnote-ref-1)