TEA Performance (May 2021)

Abstract— The rapid growth of IoT devices has formed large scale data networks that could become major targets of data theft. The security risks of IoT devices can be mitigated using encryption and decryption algorithms to protect data while it is shared. These algorithms have impacts on the performance and power draw of IoT devices that they run on and need to be documented to optimize the efficiency of security applications. This paper describes the mechanics of the algorithm and analyzes multiple performance metrics and the power consumption of the Tiny Encryption Algorithm on a Raspberry Pi 3B+ computer. Specifically, a comparison is made between the performance metrics of the Python implementation and C++ implementation of the algorithm. The data collection methodology is also made available. The results of the study shows that the C++ implementation of the Tiny Encryption Algorithm runs 11.7 times faster than the Python implementation on a Raspberry Pi 3B+ running a 32-bit Raspberry Pi OS. The C++ implementation is able to complete 50 encipher-decipher pairs in 7279302 nanoseconds while the Python implementation required 84918855 nanoseconds. Comparisons between the C++ implementations of the improved Extended Tiny Encryption Algorithm and the original Tiny Encryption Algorithm shows that the improved version of the algorithm runs 8.7 times faster than the original. In terms of power consumption, a 5200 mAh power bank supplying the Raspberry Pi 3B+ yielded a maximum run time of 13.7 hours. Additional power estimations are conducted using three 18650 batteries connected in series as a basis.

Sydney Cady, Ryan Schwartz, Steven Yu, Mohamed El-Hadedy

California State Polytechnic University, Pomona

College of Engineering

Department of Electrical and Computer Engineering

rmschwartz@cpp.edu, slcady@cpp.edu, stevenyu@cpp.edu, **mealy@cpp.edu**

**Index Terms—Tiny Encryption, Decryption, TEA, Raspberry Pi, ARM, Benchmark, Internet of Things, IoT, Security, Smart Device, Cryptography**

# INTRODUCTION

The networks of IoT devices are growing at an exponential rate with no signs of slowing down. The global market for IoT devices reached 100 billion for the first time in 2017 and is projected to grow to a staggering 1.6 trillion by 2025 [4]. This rapid growth is driven by an increasing public interest in the advantages and conveniences that are offered by interconnected devices. Smart devices such as Apple and Android phones as well as the Amazon Echo and Google Home are now common place pockets and in homes across America. They make it possible for its users to operate any number of connected devices by delivering simple voice commands. Smart homes that unlock doors with facial recognition, turn on ambient music with voice commands, and activate motorized window shutters with a tablet have become a reality of the present rather than a dream of the future. The tone and color of Panasonic light bulbs can now be changed through a swipe on a tablet, Samsung refrigerators are fitted with cameras that display the contents of the refrigerator on a phone without needing to check inside, Ring security cameras turn any phone or tablet into a security monitoring system, and the possibilities are endless. As the potential features and benefits expand the list of interconnected devices also follows suit to form expansive networks of data in every household.

However, the rapid growth of these data networks have also gained the unwanted notice of hackers and data miners who are interested in stealing the masses of information made available by the many sensors of IoT devices that are now installed in these homes. Thus, the growth of IoT device networks also calls into concern the threat of data leaks and hacking which data security and data encryption to the forefront of the discussion. Data encryption algorithms like the Tiny Encryption Algorithm that prevent private data from being read by those with malicious intentions is crucial for maintaining the integrity of IoT systems. It is important for these algorithms to be able to efficiently encrypt data before it is sent and rapidly decrypt data that is received while minimizing performance loss on the IoT device that it operates on. In order to study the impacts of data encryption and decryption on such devices, this paper develops a study and documents various performance metrics of the Tiny Encryption Algorithm executed on an ARM Cortex-A53 based Raspberry Pi 3B+ computer. The study is conducted by comparing the performance data yielded by a Python implementation of the encryption algorithm and a C++ implementation of the algorithm. A comparison is also made between the improved Extended Tiny Encryption Algorithm and the original Tiny Encryption Algorithm in C++ using Amdahl’s Law. The performance metrics that are collected and analyzed include execution time, CPU rate, cycles per second, MIPS, CPI, and additional power consumption measurements. Power consumption measurements include analysis using a power meter as well as additional power estimations.

The next sections of this paper is presented as follows. Section 2 of this paper shows the various related works. Section 3 provides background information on the objective of this research. Section 4 gives a brief overview on the mechanics of TEA. Section 5 displays the collected performance data of running TEA on a Raspberry Pi 3B+. Section 6 details the methodology of the data collection and analysis of the results. Section 7 will then conclude this paper.

# Related Works

In [5] the author opted to measure the power consumption of the Raspberry Pi using a power bank. The power study in this paper collected data from both a power bank and individual lithium batteries connected in parallel. This change was made because the power indicator on the power bank does not give accurate readings and the individual batteries allowed for voltage measurements of the batteries to be made.

[6] presents a comprehensive perspective of TEA and XTEA and analyzes the severity of their weaknesses to cryptanalytic attacks. It finds that the encryption of cipher texts with less than six Feistel rounds are significantly weaker to cryptanalytic attacks than ones with more than six rounds. Encryption using more than six rounds showed a better mixture of intermediate values and a higher resistance to the attacks. In order maintain a higher degree of security when encrypting private information, the implementation of TEA in this report uses 64 Feistel rounds in the encryption process. This change is likely to increase memory usage and power consumption in small devices, however the increase in security is crucial for preventing data leaks and for maintaining the trust of its users.

[7] studies the resource utilization of the Blowfish, AES, 3DES and DES encryption algorithms. Its results show that the key length of the cryptographic algorithm is proportional to the amount of resources that it utilizes. These four algorithms allow the use of key sizes larger than the 128 bit key found in the TEA. Such changes likely increase their defensive capabilities against attacks at the cost of higher resource utilization and slower execution times when compared to TEA. This suggests that although increasing the size of the key would improve the security of the encryption, it is not recommended to drastically increase the key size because the longer encryption process consumes more memory and power while producing excess heat. Striking a balance between security and resource consumption is crucial especially in memory sensitive applications that the TEA is designed for.

# Background

Most of the understanding comes from understanding the few lectures within computer architecture class, which brought understandings of metrics within computer performance. By using this information, one can understand how programs function and the measurements that can be taken. Within this project, a Raspberry Pi 3B+ and use Python and C++ are used in conjunction with the TEA encryption. This paper measures the metrics of each language and compare it to one another. This is done through timers within the system and calculations with the data found in each program. 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 goes over the understanding of TEA through explanation and graphics, then explains the execution of the TEA algorithm in Python and C++.

## 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 is 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.

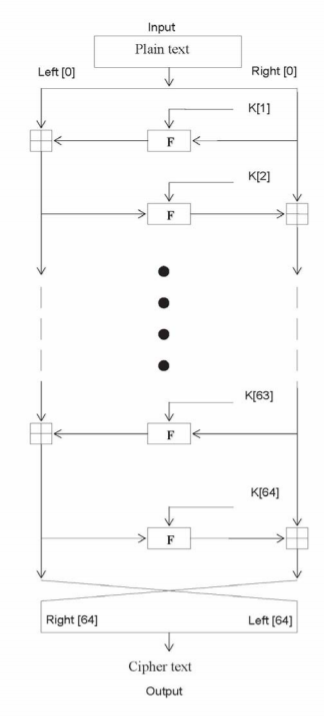


Fig 1. Here is an abstract structure of the TEA routine. As you see it go through about 64 cycles before reaching the end.

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

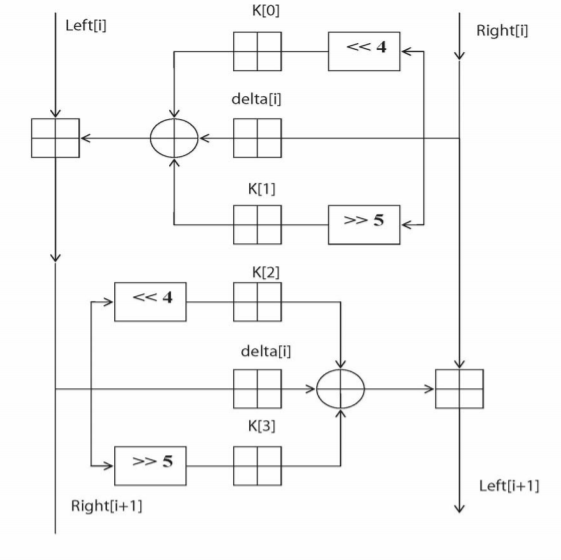


Fig. 2. This is an abstract view of the i-th iteration of the TEA cycle.

To understand what these last two figures represent, one must first understand what each notation means. Each round (i) has 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

Here is the code used to take measurements for the experiment. The code used in Python is displayed below:

|  |
| --- |
| import sys  from ctypes import \*  import time  def encipher(v, k):  start counter for timing  y = 32-bit int  z = 32-bit int  sum = 32-bit int  delta = 0x9e3779b9 constant  w = [ , ] result pair  for 32 loops:  sum += delta  y += shifting/XOR calculation  z += shifting/XOR calculation  w[0] = y  w[1] = z  end counter for timing  return w  def decipher(v, k):  start counter for timing  y = 32-bit int  z = 23-bit int  sum = 0xc6ef3720 constant  delta = 0x9e3779b9 constant  w = [ , ] result pair  for 32 loops  z -= shifting/XOR calculation  y -= shifting/XOR calculation  sum -= delta  w[0] = y  w[1] = z  end counter for timing  return w  def encipher\_decipher(v, key):  decipher(encipher(v, key), key)  if \_\_name\_\_ == "\_\_main\_\_":  star counter for timing  key = [ , , , ] random number key array  v = [ , ] random pair  encipher\_decipher(v, key)  end counter for timing |

In order to get the instruction count, the Python code has to be converted into assembly. This was done through using a website called godbolt.org. The website gave a *rough* value; 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, z, sum = 32-bit int  delta=0x9E3779B9  for 32 loops:  {  sum += delta;  y += shifting/XOR calculation  z += shifting/XOR calculation  }  w[0]=y; w[1]=z;  }  void decipher(unsigned long \*const v, unsigned long \*const w,  const unsigned long \*const k)  {  unsigned long y, z = 32-bit int  sum=0xC6EF3720  delta=0x9E3779B9  for 32 loops:  {  z -= shifting/XOR calculation  y -= shifting/XOR calculation  sum -= delta;  }  w[0]=y; w[1]=z;  }  int main()  {  Start counter for timing  unsigned long v[] = { , } random array pair  unsigned long key[] = { , , , } random number key array  unsigned long res[2] size of result  encipher(v, res, key);  decipher(v, res, key);  stop counter for timing  return 0;  } |

In order to get the instruction count, the C++ code must be converted 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. 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

Now this section goes into the different metrics found. Here are tables of all the metrics 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 | 0.0016983771 s |
| CPU clock cycles | 2038.13 |
| CPI | 5.26629 |
| MIPS | 0.227865 |

Table 2. Python metrics collected with TEA on Raspberry Pi

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

|  |  |
| --- | --- |
| C++ Data | Value |
| Assembly instruction count | 687 |
| CPU/Execution time | 0.000145586 s |
| 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 goes further into within the next sections in order to understand the math behind some of the performance metrics.

# TEA Data Collection

This section reviews: Clock cycles, CPI, and MIPS. The equations used to determine these metrics are shown throughout this section.

## CPU Clock Cycles

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

The basic equation used:

Thus, for Python, by substituting the CPU execution time and clock cycle time from the python and Raspberry Pi tables. Which gives us:

Then with C++, using the same equation in order to determine the CPU clock cycles.

## CPI

Then, from the data found, CPU, in conjunction with other values the CPI is found for C++ and Python. This is done through the CPI equation as shown below:

This equation is used to determine the CPI for Python and C++. First let’s determine the CPI for Python:

Then for C++ it is determined that the CPI is:

## MIPS

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

Through this equation the MIPS for the Python code can be found:

0.227865

Then by applying the same logic to C++, 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 using a USB power meter.

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.

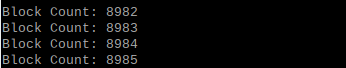


Fig. 3. Block Count From One TEA Process (12 minutes)

|  |  |  |  |
| --- | --- | --- | --- |
| Time (min.) | Voltage (V) | Current (A) | Power (Wh) |
| 2 | 4.75 | 0.34 | 0.05 |
| 4 | 4.83 | 0.34 | 0.11 |
| 6 | 4.84 | 0.4 | 0.17 |
| 8 | 4.84 | 0.41 | 0.23 |
| 10 | 4.85 | 0.41 | 0.29 |
| 12 | 4.83 | 0.39 | 0.35 |

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 | 0.05 |
| 4 | 149750 | 0.11 |
| 6 | 224625 | 0.17 |
| 8 | 299500 | 0.23 |
| 10 | 374375 | 0.29 |
| 12 | 449250 | 0.35 |

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.

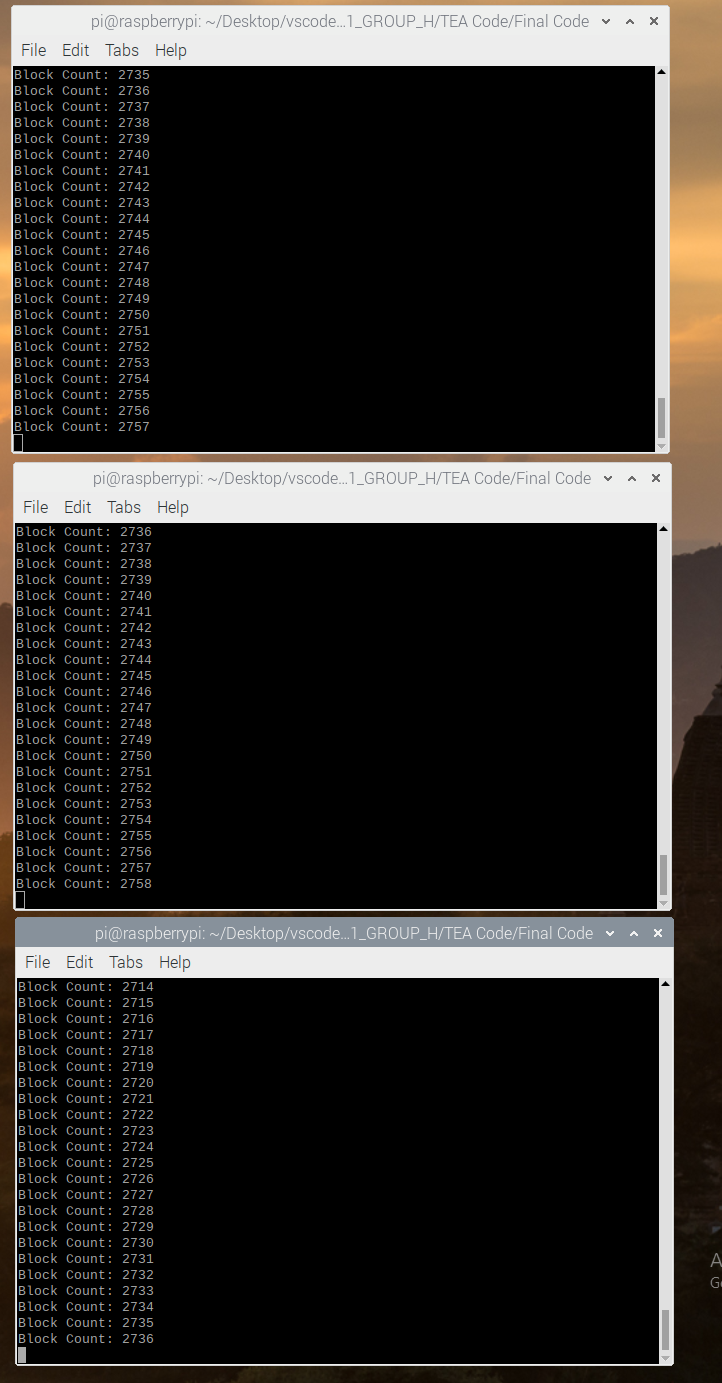








Fig. 10.Three Block Counts From Parallel Execution (12 minutes)

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 | 0.53 | 0.08 |
| 4 | 4.7 | 0.48 | 0.16 |
| 6 | 4.69 | 0.54 | 0.24 |
| 8 | 4.72 | 0.54 | 0.33 |
| 10 | 4.62 | 0.54 | 0.41 |
| 12 | 4.74 | 0.58 | 0.49 |

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 | 0.08 |
| 4 | 397116.7 | 0.16 |
| 6 | 595565 | 0.24 |
| 8 | 794123.3 | 0.33 |
| 10 | 992681.7 | 0.41 |
| 12 | 1191350 | 0.49 |

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

Shown below is all the photos taken with the Power Meter to show the data 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 3B+ 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 linear 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, it should see the power bank at half charge (9.6 hours / 4.8 hours = 2). This is the final decision of the team because of the more reasonable requirement of sitting at a desk and making sure the power bank continues to deliver power to the Raspberry Pi during the testing, 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 remaining in the power bank could be anywhere between 65% to 40%, for example. Nevertheless, the theory is correct with the data shown by the portable power bank.

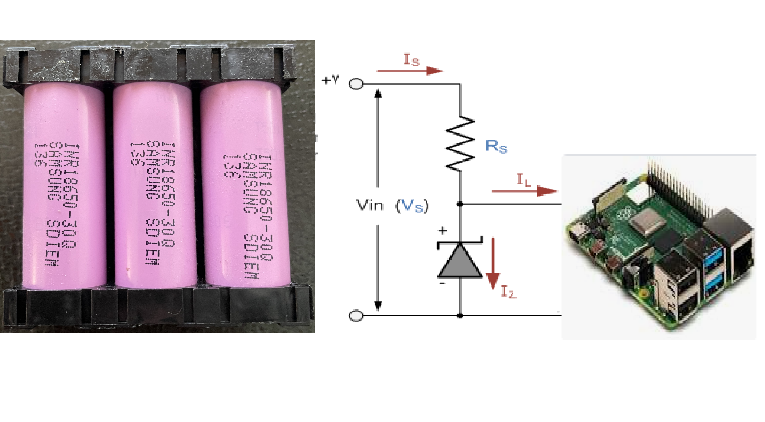
Instead of an expensive power bank, it is theorized by the team that an individual could power the Raspberry Pi 3B+ using multiple of the popular 18650 battery. 18650 batteries are the standard in today’s market when it comes to rechargeable devices, so it may be more likely for an average household 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 3.7V 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.

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

Using the data found in this report, the fraction would be new/old = XTEA assembly line count / TEA assembly line count. Then, to find S, the speedup of enhanced fraction, the value from before where it explains how much faster the program runs is:

Thus, the overall speedup is 1.03036 from these calculations. It is proven 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, the team was 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, it was found 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 there was an attempt to break down the python code into assembly. The Python resource stated that the Python code equated to about 400 lines of assembly even though it may have been more or less. This is inferred 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 the paper used this amount of instructions as a metric for the data, it is unsure as to how accurate it is because the resources are inaccurate.

As stated above, it was concluded that the C++ code runs nearly 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 the data. For example, it is understood 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 when analyzing the MIPS (million instruction per second) with the understanding of the execution time. For example, with the faster run time, it is inferred that the MIPS is pretty high. For C++, it is calculated that the MIPS received a fairly high value. Whereas when the system ran slower it 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, a better understanding of how TEA performs with power consumption was evaluated. 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 the 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, it is shown that the C++ program is faster than Python by a factor of 11.67 on 32-bit Raspberry Pi OS.

Benchmarking Tools

* Raspberry Pi 3B+
* Python GodBolt Compiler Explorer​
* C++ GNU GCC compiler​
* Python Time module
* C++ Chrono library
* USB power meter
* TYLT WA-15TPP5200 5200 mAh power bank
* Samsung 30Q 18650 rechargeable batteries

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)
4. L. S. Vailshery, “IoT market size worldwide 2017-2025,” Statista, 22-Jan-2021. Available: https://www.statista.com/statistics/976313/global-iot-market-size/.
5. “Pi duration tests and review of two new lithium battery packs,” RasPi.TV, 19-Jun-2013. [Online]. Available: https://raspi.tv/2013/pi-duration-tests-and-review-of-two-new-lithium-battery-packs.
6. V. R. Andem, “A Cryptanalysis of the Tiny Encryption Algorithm,” thesis, 2003.
7. J. B. Awotunde, A. O. Ameen, I. D. Oladipo, A. R. Tomori, and M. Abdulraheem, “Evaluation of four encryption algorithms for viability, reliability and performance estimation,” Nigerian Journal of Technological Development, vol. 13, no. 2, p. 74, 2017.

Biography

**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.