Developing CyberSecurity Skills in Intermediate Programming Courses

Jeffrey L. Duffany[1], Ph.D.
Ana G. Mendez University, Puerto Rico, U.S.A., *jeduffany@suagm.edu*

Abstract– ***For most electrical and computer engineering students intermediate programming is a required course. Typical topics taught in intermediate programming are advanced array operations, reading data from files, writing data to files, string conversions, string manipulations, pointers, searching for a text string, sorting of data into alphabetical order and numerical methods. To motivate students instructors should create learning experiences that the student can relate to for example by incorporating popular culture into the teaching materials. Judicious choice of teaching material can spell the difference between engaging or not engaging the interest of the student. One method for improving student engagement would be to provide evidence that they are learning a skill that will in the long run be useful to them in some way. Many cybersecurity concepts can be introduced and learned using techniques suitable for intermediate programming classes.***
Keywords-- cybersecurity, programming, capture the flag

I. Introduction

Traditional learning environments are largely passive in nature and that can actually be an efficient way of delivering a lot of material[1]. However recently there has been a trend towards what is called "active learning"[1] where the passive teaching techniques are reduced to a minimum while the student is more "actively" engaged in the learning process through a variety of techniques. However for active learning to work the student must be motivated to learn in the first place[1][2].

Programming classes typically have a lecture/lab format where the class is given in computer lab environment where each student has their own computer or in some cases the course is divided into separate lecture and lab components. Either way a hands-on programming experience is an integral part of the course and as such the requisite facilities in terms of hardware and software must be provided in the classroom environment. This paper describes several ideas for introducing cybersecurity concepts[3][8] to undergraduate engineering students taking intermediate programming. There is nothing inherently wrong with the current curriculum approach. The suggestion is to teach the same material in a context that will help to increase the motivation of the students.

The concept of cyber security has been filtering into the public conscious for quite some time as evidenced by the fact that there are stories about computer hackers in the news practically every day[3]. The US Government Agencies have a large vacuum of cyber-security-related jobs to fill and there are currently not enough skilled people with training[3]. Cybersecurity has long been the domain of graduate studies however it is starting to trickle down to the undergraduate level[3][6][8]. GenCyber is (a.k.a Generation Cyber) is a joint initiative by the NSF and NSA to introduce cybersecurity topics to K-12 students[7]. This is often done in a summer camp environment that introduces the concepts in a very broad and generic way. The camps are free of charge and take place every summer across the United States and Puerto Rico[7].

II. Background and Motivation

One way to improve motivation is to illustrate tangible benefits of performing some activity (such as for example learning or sports training)[2]. Another way is to inspire the students by somehow capturing their imagination which can lead to an inner self-motivation[4][5]. As a case in point there is nothing quite like a competition to motivate people; as a classic example an entire industry of sports is built on that principle. Imagine playing a game of ping pong or tennis without keeping track of points. What will inevitably wind up happening is that you will end up just hitting the ball around back and forth amiably. Now imagine what happens when both players agree to keep score and they decide that first player to score say for example 11 points wins. The result is that it will invariably lead to a fundamental change in how the players interact with each other as keeping score is both literally and figuratively a game changer. It will often bring out the best in people. When not keeping score players tend to have less focus and or a different focus. For example instead trying to win every point the players may choose to focus on practicing their topspin, backspin or sidespin techniques instead.

So how does that previous sports discussion relate to intermediate programming? In recent years the concept of cybersecurity competition, e.g., capture the flag (CTF) has been gaining in popularity[6]. A premier example is the National Cyber League (NCL)[10]. The National Cyber League is a an online competition that is geared towards high school and university students. The competition is held twice a year once in the spring and the fall. Last year over 5000 students participated in the NCL Fall Season individual competition[10]. Every person who enters becomes nationally ranked according to their standing and the highest scoring individuals are placed gold, silver or bronze status. All of the NCL challenges are geared towards helping the student gain a deeper understanding of cybersecurity concepts and principles. It is this ranking system that motivates the participants to solve as many challenges as possible.

Digital Object Identifier (DOI):
http://dx.doi.org/10.18687/LACCEI2019.1.1.414
ISBN: 978-0-9993443-6-1 ISSN: 2414-6390

There are many other CTF and CTF-like competitions sponsored at conferences, by universities and other organizations (recently a CTF competition was held at the Grace Hopper conference for women in computing sciences[17]). The National Collegiate Cyber Defense Competition (CCDC) is another well-known CTF event for university students. Another potential source of inspiration is from literature and popular culture (criminal history) and there are a number of famous cryptographic messages some of which have been solved and some which have not. Well-known examples include Arthur Conan Doyle "Sherlock Holmes Adventure of the Dancing Men" [9], Edgar Allan Poe "The Gold Bug", the Zodiac Killer Ciphers[18] and the DaVinci Code by Dan Brown[4].

Figure 1. Kryptos Sculpture at CIA Headquarters

Perhaps one of the most famous examples of unsolved cryptographic code[5] can be found etched into a statue at CIA headquarters in Langley, Virginia (Figure 1). In 1988, as a new headquarters for the American Central Intelligence Agency (CIA) was being built and sculptor Jim Sanborn was commissioned to create artwork for the courtyard of the new building. He designed a large copper monument, shaped somewhat like a flag, and engraved with an encrypted message. The name of the sculpture is *Kryptos[5]*. Now over 20 years after the statue was unveiled the encrypted text has yet to be fully deciphered.

Kryptos has seen three of its four sections solved however still uncracked are the 97 characters of the fourth part (see Figure 1). The combined efforts of the NSA and CIA - some of the world's top crypto experts - have been unable to break the code. Imagine the inspiration this could instill in high school or college student knowing how famous they would become if they could crack the infamous (and as of today still unsolved) fourth part of the Kryptos code[5].

III. Intermediate Programming Curriculum

The standard curriculum of intermediate programming courses typically include: advanced array operations, reading data from files, writing data to files, string conversions, string manipulations, pointers, searching, sorting of data and numerical methods[[13]. It will be shown in this paper that intermediate programming techniques coincide very well with the cybersecurity programming tasks needed for CTF competitions[13].

IV. Illustrative Examples

The main idea is to show students how easily written and relatively short computer programs can be used to help them solve cybersecurity related challenges. This allows the instructor to teach the concepts of intermediate programming and cybersecurity at the same time. The following topics have been chosen as illustrative and will be discussed in detail throughout the remainder of this paper.

A. Cryptography - Caesar Cipher
B. Brute force Cryptanalysis
C. Cryptography - Railfence
D. Frequency Analysis
E. Password Cracking

These examples were chosen to be easy to program and at the same time provide tangible evidence to illustrate how programming skills are likely to benefit the student later on. There are many other examples that could have been chosen so these should be considered mainly as representative. These examples are based directly on actual challenges found in the National Cyber League and other CTF competitions[6][10]. These programs can be used "as is" to solve a number of the easier challenges and can be modified for more difficult ones.

More difficult challenges will require writing of new programs however they will employ many of the same tools and techniques illustrated here. All of the programs included in this paper are fully working computer programs and not merely just pseudocode. They are intended to facilitate anyone reading this paper to help to adapt it to their particular curriculum. They are written in R Language[12] but can be rewritten in any of the popular teaching languages such as C++[13], Python[15], Visual Basic[14], etc. R is a programming language environment that is free to download and use[12]. R is an interpreted language that does not need a compiler which makes it easy to quickly write programs and get them working with minimum knowledge or expertise. Programs are written by taking commands and storing them in a file and then submitting it to a "shell" type of environment which sequentially executes the commands in the file. All standard programming constructs such as repetition loops and "if" statements are fully supported.

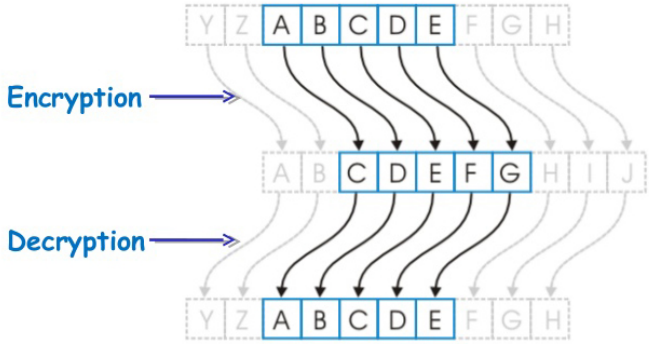

FIGURE 2 CAESAR SHIFT CIPHER (SHIFT= 2)

A. CRYPTOGRAPHY - CAESAR SHIFT CIPHER

An example of classical cryptography is the caesar cipher also known as the caesar shift cipher[9] which is known to almost every schoolboy and girl as the "secret decoder ring". The idea of the Caesar shift cipher is illustrated in Figure 2. It is a classical substitution code which replaces letters of the alphabet with other letters to the left or right as a method of obfuscating the "top secret" message. For this discussion the letter capital "P" represents the unencrypted original or "plaintext" and capital letter "C" represents encrypted ciphertext. The Caesar cipher is represented by a mathematical formula shown in Equations (1) and (2) below with the key to encrypt or decrypt given by an integer K which represents the amount of the shift. In order for equations (1) and (2) to work properly the letters of the alphabet must first be assigned numbers starting with A=0, B=1,....ending with Z=25.

$$C = (P + K) \bmod 26 \quad \text{(encryption)} \quad (1)$$

$$P = (C - K) \bmod 26 \quad \text{(decryption)} \quad (2)$$

A simple computer program for implementing the caesar shift cipher is shown in Figure 3. The program was written in R Language[12]. There is nothing special about R language and many other languages could have been used[13][14][15]. R language was used because R is a high level language and the programs tend to be shorter thus making them elegantly suitable for illustration. So if the input plaintext is P = "buenos aires" and the amount of shift is one letter to the right then the output would be "cvfopt bjsft"

In this case it can be seen that a key value K=1 was used in Equation (1). Using a shift K=1 the letter "a" becomes "b" and the letter "b" becomes "c". This is evident in the ciphertext as it easily seen that the c in "cvfopt" is the shifted b from "buenos" and the b in "bjsft" is the shifted a from "aires". Adding an integer K is a shift to the right in the alphabet while subtracting an integer K is a shift to the left in the alphabet.

```
function caesar_encrypt()
{
plaintext<-"buenos aires"
k=1
print(plaintext)
ascii<-utf8ToInt(plaintext)-97
space<-ascii<0
ciphertext<-(ascii+k)%%26+97
ciphertext[space]<-32
ciphertext<-intToUtf8(ciphertext)
print(ciphertext)
}
```

Figure 3. Caesar Shift Cipher Encryption in R Language

To understand how the R Language program in Figure 3 implements Eq.(1) a little background explanation is required. Information is represented inside a computer in ASCII code[19], UNICODE[20] or UTF-8[21]. In all of these coding systems the letter "A" is represented as decimal number 65 and correspondingly that means B=66, C=67, etc. The letter Z is represented by decimal number 90. The lowercase letters are a=97 to z=122. Since the caesar cipher encryption program in Figure 3 is designed to work only with lowercase letters (for simplicity) it is necessary to subtract 97 at the beginning of program after utf8 encoding in line 4. Eq. (1) is implemented line 6 of the program ciphertext<-(ascii+k)%%26+97 and the result shifted back up by adding 97. The double percent %% is the modular division operator in R Language which implements the mod function from Eq. (1). The mod (modulo) operator is the remainder after integer division for example 26%%26 =0 (26 divided by 26 =1 with no remainder as 26 goes into 26 exactly once). So if K=26 then A+26= A+0 since 26%%26=0. Therefore if the key K=0 the encryption algorithm will return a ciphertext = plaintext so keys of either 0 or 26 are null keys. In line 7 space<-ascii<0 finds the location of all spaces as 32 is the ASCII code for space " ". Line 7 ciphertext[space] <-32 puts back the spaces (i.e., the space between "buenos" and "aires").

B. Cryptanalysis

While cryptography is the art of making codes cryptanalysis is the science of breaking codes[9]. In other words cryptanalysis represents the tools and techniques for finding the plaintext (P) given only the ciphertext (C). When the key (K) is known to the message recipient decryption is simple for example in the case of the caesar cipher using the formula in Equation 2. The key K is subtracted from each letter in the message shifting to the left thereby recovering the plaintext.

When the key is not known the process of finding the message plaintext is called "cryptanalysis". There are two primary techniques used in cryptanalysis: brute force and frequency analysis[9]. The brute force technique tries every possible key while the other methods of cryptanalysis try to impose a maximum likelihood ordering on the keyspace by trying the most likely keys first. Figure 4 shows an R Language program for performing cryptanalysis of the caesar cipher using the brute force technique.

```
function caesar_decrypt()
{
ciphertext="leoxyc ksboc"
for(k in 0:25) {
ascii<-utf8ToInt(ciphertext)-97
space<-ascii<0
plaintext<-(ascii-k)%%26+97
plaintext[space]<-32
plaintext<-intToUtf8(plaintext)
print(plaintext)
}}
```

Figure 4. Cryptanalysis of the Caesar shift cipher

Just as with the encryption program in Figure 3 the decryption program first changes the letters to integers and then subtracts 97 to set the letter "a" to zero. It then subtracts the key which is the inverse of adding the key during the encryption process.

Suppose for example the captured secret message was C= ciphertext = "leoxyc ksboc". This represents a caesar cipher of shift (or key) K = 10. As can be seen in the ciphertext the "b" in buenos aires becomes "l" in "leoxyc ksboc" since "b" is the second letter of the alphabet and "l" is the twelfth letter of the alphabet "b"+K="l" (i.e., 2 +10 = 12).

The full result of the brute force cryptoanalysis is shown in Figure 5 which is the output of the program in Figure 4. The program tries every key starting with a key=0 and going to 25 as seen in the repetition loop. This clearly illustrates the "needle in the haystack" type of search that typically results from a brute force cryptanalysis.

As seen in Figure 5 the first key (K=0) is a null key which just returns the ciphertext. Finally when the key K=10 is tried the plaintext P = "buenos aires" emerges. This approach does not scale very well for the large keyspaces of modern cryptography.

```
[0] "leoxyc ksboc" Ciphertext
[1] "kdnwxb jranb"
[2] "jcmvwa iqzma"
[3] "ibluvz hpylz"
[4] "haktuy goxky"
[5] "gzjstx fnwjx"
[6] "fyirsw emviw"
[7] "exhqrv dluhv"
[8] "dwgpqu cktgu"
[9] "cvfopt bjsft"
[10] "buenos aires" plaintext
[11] "atdmnr zhqdr"
[12] "zsclmq ygpcq"
[13] "yrbklp xfobp"
[14] "xqajko wenao"
[15] "wpzijn vdmzn"
[16] "voyhim uclym"
[17] "unxghl tbkxl"
[18] "tmwfgk sajwk"
[19] "slvefj rzivj"
[20] "rkudei qyhui"
[21] "qjtcdh pxgth"
[22] "pisbcg owfsg"
[23] "ohrabf nverf"
[24] "ngqzae mudqe"
[25] "mfpyzd ltcpd"
```

Figure 5. Brute Force Cryptoanalysis of the Caesar Cipher

This technique can work in cases where the keyspace is small but does not scale very well to cryptography for large keyspaces[9]. At this point the students can be given a challenge assignment whereby they have to figure out an automatic way of detecting when the key is found. There are number of ways of doing this most of which depend on dictionary lookup or spell check techniques. The purpose of this challenge is to stimulate the student's imagination and curiosity and create motivation for wanting to learn more about computer programming. It also encourages further exploration by the student. For example what other techniques can be used to automatically detect when the key has been found? What happens when you make a slight modification to create a monoalphabetic substitution code increase the keyspace to 400 trillion trillion (keys). These ideas can be explored without a disproportionate amount of effort to modify the code.

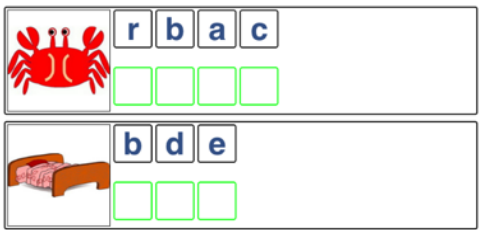

Figure 6. Some Simple Transposition Ciphers

C. TRANSPOSITION CIPHERS - RAILFENCE

Transposition ciphers (permutation ciphers)[9] do not use substitution instead they simply change the ordering of the letters in the message. This is done in some systematic way that is previously negotiated by the sender and recipient. Figure 6 shows how transposing the order of the characters in plaintext can result in obfuscation of the original message. These codes by themselves are easy to solve using brute force of all possible permutations and it is readily seen that the solutions are "crab" and "bed". The solution is shown pictorially in Figure 6 as a means of reinforcing that the correct solution was found or perhaps as a hint. By themselves transposition codes are not very secure as they can easily be solved with frequency analysis and brute force techniques. However when combined with substitution codes they form the basis for product codes which are the most powerful cryptographic methods known[9].

Perhaps the most well known example of a transposition cipher is the rail fence. In the rail fence cipher the plain text is written alternatively on successive "rails" of an imaginary fence. When the end of the message is reached it is read off the rails in rows. For example consider the plaintext string: "me gusta la gasolina dame mas gasolina" or "megustalagasolinadamemasgasolina" after removing the spaces. Putting the letters of the plaintext into the railfence gives the result in Figure 7.

(a) P = ME GUSTA LA GASOLINA DAME MAS GASOLINA

(b) P = MEGUSTALAGASOLINADAMEMASGASOLINA

(c) MGSAAAOIAAEAGSLN (RAIL 1)
EUTLGSLNDMMSAOIA (RAIL 2)

(d) C = MGSAAAOIAAEAGSLNEUTLGSLNDMMSAOIA

Figure 7. Illustration of the Railfence Cipher

Figure 7(a) shows the original plaintext (P) and Figure 7(b) shows the plaintext with spaces removed. In Figure 7(c) the letters of the plaintext are placed alternatively on two rails - first the M on RAIL 1 and then the E on RAIL 2, etc. In Figure 7(d) the second rail is concatenated at the end of the first rail resulting in the ciphertext (C) shown. The R Language code for implementing a railfence cipher is given in Figure 8.

```
FUNCTION RAILFENCE()
{
PLAINTEXT="MEGUSTALAGASOLINADAMEMASGASOLINA"
ASCII<-UTF8TOINT(PLAINTEXT)
LEN<-LENGTH(ASCII)
K=2
J=0:(LEN-1)
J<-J%%K
CIPHERTEXT<-NULL
FOR (I IN 0:(K-1)) {
CIPHERTEXT<-C(CIPHERTEXT,ASCII[J==I])}
CIPHERTEXT=INTTOUTF8(CIPHERTEXT)
PRINT(CIPHERTEXT)}
```

FIGURE 8. R LANGUAGE PROGRAM FOR THE RAILFENCE CIPHER

The R Language program in Figure 8 works as follows. The plaintext input is first converted into the integer form of ASCII code where the letter A=65, B=66, etc. Next the length of the plaintext string is computed and the value of the key is specified in this case K=2. A value of K=2 corresponds to 2 rails in the railfence as illustrated in Figure 7. Each plaintext letter is then assigned a unique integer value from 0 to the length of the plaintext string minus 1. This vector is then divided by the number of rails K (modulo K) which assigns each letter to a rail.

The railfence technique is very similar to the method of assigning students into groups by counting off numbers (1 2 1 2 1 2) for two groups and (1 2 3 1 2 3) for 3 groups, etc. used for example to place students who are friends with each other in different groups. The R Language railfence program works the same way. A vector index is created according to the key and the length of the ciphertext string. In this case it would be 01010101010101... (the binary equivalent of "boy-girl-boy-girl" seating arrangements). All plaintext letters of index=0 are extracted from the plaintext then all letters of index=1 and the two strings are concatenated together resulting in the ciphertext string C= "mgsaaaoiaaeagslneutlgslndmmsaoia".

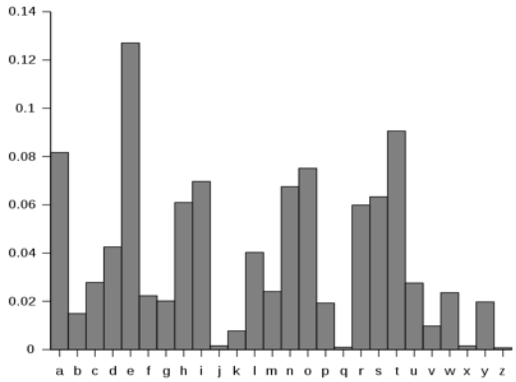

Figure 9. Frequency of Letters in the English Language

D. Letter Frequency Analysis

In cryptanalysis, letter frequency analysis is the study of the individual letters or groups of letters in a ciphertext[9]. The method is used as an aid to breaking classical cipher. Frequency analysis is based on the fact that, in any given sample of written language, certain letters and combinations of letters occur with different frequencies. For instance, given a sample of English language, E, T, A and O are the most common letters while Z, Q and X are least common. The 12 most common letters in typical English language text are: ETAOINSHRDLU with E being the most common letter at 12.8%. An R Language program for letter frequency analysis is given in Figure 10.

```
function letter_frequency_analysis()
{
ciphertext="leoxy cksboc"
ascii<-utf8ToInt(ciphertext)
hist(ascii, breaks = (97:123)-.5)
count<- hist(ascii, breaks = (97:123)-.5, plot = FALSE)
print(ciphertext)
print(count)
}
```

Figure 10. R Language Program For Letter Frequency Analysis

The R language program in Figure 10 makes use of the built-in histogram function hist(). However just using hist() by itself will not give the desired results you have to specify the breaks with a granularity that includes exactly one letter in each bar in the resulting histogram by specifying: breaks = (97:123)-.5 which is a vector = (96.5, 97.5, 98.5,.....122.5). So the first bin of the histogram will count the number of 97s (a's), and the second bin will count the number of b's, etc.

Frequency analysis is a commonly used tool in cryptanalysis[9]. To use it to break a caesar cipher one performs a frequency analysis of the ciphertext and compares it with the frequency graphic in Figure 9. The letter in the ciphertext that has the highest frequency is likely to be an "e" in the plaintext. Based on that information you guess the key matching the most frequent characters in the ciphertext to the most frequent characters in the English (or whatever) language. Figure 11 shows the result of the frequency analysis program on the ciphertext C = "leoxyc ksboc".

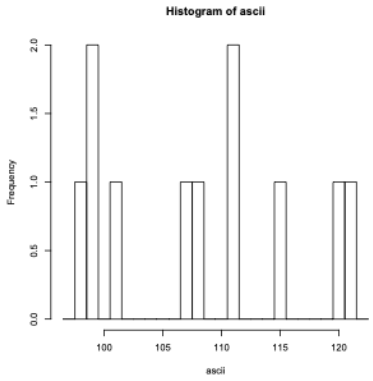

Figure 11. Letter Frequency of "leoxyc ksboc"

As can be seen from Figure 11 the most frequent letters are ascii 99 (c) and 111 (o) which corresponds to caesar cipher shifts of K=23 and K=10. Since "e" (ascii 101) is the most frequent letter in the alphabet the most likely possibility is that the encryption key K=10 since 101(e)+10=111(o). Applying a shift of -10 to ciphertext = "leoxyc ksboc" shows that our intuition was correct resulting in plaintext P = "buenos aires" as "l" =12 shifts by -10 to b=2, etc.. With letter frequency analysis the code was cracked on the first try as opposed to the brute force approach in Figure 5 which had to try all 26 keys.

Frequency analysis doesn't always work especially on such a small sample of ciphertext but it does work most of the time. For this case we were able to crack the code after one guess which is typical for a caesar cipher. However in general a more systematic method is needed whereby the frequency analysis is used to prioritize the order of the brute force search[9]. Using a brute force approach as illustrated in Figure 5 can work for small keyspaces but does not scale well for large keyspaces. For the monoalphabetic substitution cipher (a minor variation of the caesar shift cipher) it could take around 1000 years to crack the code even when testing 1 million keys per second[9]. However if letter frequency analysis is used it is possible to reduce this time to a matter of seconds or minutes[9].

Figure 12. Most Common Passwords from 2014

```
function password_crack()
{
 pwhash<-"f402749fb2d2a4f7092e67f9ee11
 a3b706a64e274d9c85fcc80b9076b4025e37"
  mydata<-scan("pw.txt", what=character(0))
  password<-as.matrix(mydata)
  hashval<-NULL
  n<-nrow(password)
  for (i in 1:n){
        tmp <- hash(charToRaw(password[i]))
        hashval <- bin2hex(tmp)
   if(hashval==pwhash)
   print(c("success: password=",password[i]))
 }}
```

Figure 13. R Language Program to Crack Passwords

E. PASSWORD CRACKING

Passwords are not stored on most computers for security reasons[9]. The reason is because if someone accessed your computer over a network and stole your password file they would easily be able to break into your computer. To prevent this from happening a password hash is usually stored instead of the password itself[9]. A cryptographic hash of a text string or a file is like a fingerprint of that data[9]. A hash is also a one-way function in that it does not have an inverse that can be computed directly[9].

If someone steals your password file they will have the hash of your password and there is no simple way they can calculate your password from the hash. The brute-force method to get the password is to hash every possible password until you find a match. Figure 12 shows the most common passwords from hacked websites. Skull security[16] maintains a website where many of these cracked password lists (for example "rockyou") can be downloaded. NCL challenges in past years have historically involved various forms of password hacking[10]. Typically you are given the hash of a password and you have to find the password. You are also given a subtle hint as to where the password might be found among all the famous hacked password lists on the internet [16]. The solution technique is to download the password list and store it in a text file for example "pw.txt". Next import the file into the R Language and compute the hash of every password on the list until you find a match. When you do find a match the password will be the password that generated the matching hash (see Figure 13).

V. Going Beyond The Basics

The R Language programs in this paper were only able to explore the basic concepts of cybersecurity. However these programs do provide a starting point for exploring more advanced concepts. For example there are many other types of cipher methods that were not discussed both substitution codes and transposition ciphers. Many of these can be implemented by modifying the code that has already been created (for example ATBASH cipher[9], ROT13 cipher[9], etc.). Once the basic concepts have been introduced the students should be given one or more challenge problems. These challenges can be explored without an overbearing amount of effort to modify the code. For example one challenge problem would be how to determine from the frequency analysis which keys are most likely to crack the code. There are many ways of doing this but the optimal method is not obvious and could easily be the basis for a Master's or Ph.D. thesis.

R is a very powerful language that provides many tools for statistical analysis and data mining[12]. R is supported by a worldwide community that provides hundreds if not thousands of add-in packages[12] most of which are of very specialized use (for example in bioinformatics) and not particularly useful for cybersecurity. However there are some add-on packages that can be very useful[12]. For example the Sodium package gives a wide variety of hash functions and utilities for public key cryptography. Use of this package or other similar utility would greatly extend the capabilities of R for NCL or other CTF challenges[10][11].

Another useful R Language package would be one that provides spell checking utilities (such as hunspell) which can be extremely useful in cryptanalysis[12]. We have already discussed two types of cryptanalysis (brute force and frequency analysis) unfortunately these techniques by themselves can be very awkward, tedious and time consuming by themselves. Students should be challenged to improve on these cryptanalytic techniques by making them more automatic. The goal is that the user should be able to enter a sample of ciphertext and the program will automatically return the corresponding plaintext without tedious manual trial and error.

```
function ()
{
library(hunspell)
words <- c("laccei", "engineering", "school")
hunspell_check(words)
}
```

```
[1] FALSE  TRUE  TRUE
```

Figure 14. Using the Hunspell Spell Checking Program

One method of doing this is with a spellchecker such as the hunspell spell checking program as illustrated in Figure 14. Figure 14 shows the hunspell algorithm processing a vector of words including "laccei", "engineering" and "school". The result is a logical vector that is TRUE if the word is spelled correctly and FALSE if spelled incorrectly. In the example "laccei" is detected as a misspelled word (FALSE) because it is not in the hunspell dictionary whereas "engineering" and "school are correctly identified as spelled correctly (TRUE).

Figure 5 illustrated that in a brute force cryptanalysis the result of every key is printed out which could print a lot of meaningless ciphertext. A much improved technique is to take each decrypted ciphertext from Figure 5 and filter it through the hunspell spelling checker as illustrated in Figure 14. If the result of hunspell is TRUE then print out the resulting text and you will find the plaintext "buenos aires" after trying the first 11 keys. Of course this approach only works if the plaintext you are looking for is written in the same language (English, Spanish, etc.) as the spellcheck dictionary being used.

There may be a few false positives as ciphertext can sometimes coincide with a valid word in English or Spanish. There can also be false negatives if the message contained a typographical error (as in the famous example of the message sent by Lt. John F. Kennedy from PT-109 during World War II)[9]. This is all part of the cryptography experience that the student can learn by solving cryptographic challenges.

V. DISCUSSION

When participating in a CTF such as NCL the competition is limited to around 48 hours therefore time is of the essence. Some challenges can be solved easily with pencil and paper and yet others can be solved with online resources for example the Simon Singh black chamber[22]. But there are more difficult challenges that cannot be solved with online calculators. In those cases it may be necessary to write a program to assist in searching for the solution. For example in a recent NCL competition a file of 10,000 signed messages was provided and the challenge was to identify which one of the 10,000 messages had a valid signature. This is typical of the "needle in the haystack" type of problems found on the NCL that cannot be solved by hand. By the end of the competition students with the highest scores will be given gold, silver or bronze status. To get into the gold bracket it is essential to be able to write computer programs to help find the solutions.

VI. SUMMARY AND CONCLUSIONS

For most electrical and computer engineering students intermediate programming is a required course. Typical topics taught are advanced array operations, reading data from files, writing data to files, string conversions, string manipulations, pointers, searching, sorting of data and numerical methods. It was observed that the topics taught in intermediate programming were very well matched with the cybersecurity programming tasks needed for CTF competitions. Long the realm of graduate courses cybersecurity is gradually beginning to trickle down to the undergraduate level.

Educators should do their best create learning experiences that the student can relate to for example by incorporating popular culture into the teaching materials. Judicious choice of teaching material can spell the difference between engaging or not engaging the interest of the student. One method for improving student engagement would be to provide evidence that they are learning a skill that will in the long run be useful to them in some way. Many cybersecurity concepts can be introduced and learned using techniques suitable for intermediate programming classes. The intent is to create learning experiences that inspire the students to be self-motivated out of their own curiosity and desire to learn.

It should be clear that there is nothing inherently wrong with the current approach to the teaching of intermediate programming. This paper presents an alternative based on the premise that it might be more stimulating to the intellectual curiosity of the student if the theme revolved around concepts in cybersecurity. Some of this material might equally be applied to some extent in an introductory first course in programming however cybersecurity makes use of advanced techniques more suitable for application in intermediate programming courses.

REFERENCES

[1] R.M. Felder and R. Brent, "Active Learning: An Introduction." ASQ Higher Education Brief, 2(4), August 2009.

[2] Yacob, A. and Saman, M., "Assessing Level of Motivation of Learning Programming Among Engineering Students", Frontiers in Education Conference (FIE) 16(3), 2006, p. 211-227.

[3] Cobb, S. (2016). Mind This Gap: Criminal Hacking and the Global Cybersecurity Skills Shortage, a Critical Analysis. Retrieved from https://www.virusbulletin.com/uploads/pdf/magazine/2016/VB2016-Cobb.pdf.

[4] Brown, Dan, The Da Vinci Code (1st ed.), US: Doubleday, April 2003, ISBN 0-385-50420-9.

[5] Dunin, Elonka (2009). "Kryptos: The Unsolved Enigma". In Daniel Burstein & Arne de Keijzer (editors). ISBN 978-0-06-196495-4.

[6] Leune, K., & Petrilli, S. J. 2017). Using Capture-the-Flag to Enhance the Effectiveness of Cybersecurity Education. Proceedings of the 18th Annual Conference on Information Technology Education - SIGITE 17.

[7] "NSA's GenCyber Reaches New Territories": https://www.nsa.gov/news-features/press-room/Article/1618775/nsas-gencyber-reaches-new-territories/

[8] Tobey, D. H., Pusey, P., & Burley, D. L. (2014). Engaging learners in cybersecurity careers. ACM Inroads, 5(1), 53-56.

[9] Stallings, W. "Cryptography and Network Security", ISBN 978-0-13-609704-4, Fifth Edition, Prentice Hall, 2011.

[10] National CyberLeague Regular Season: https://www.nationalcyberleague.org/regular-season

[11] National Collegiate Cyber Defense Competition: http://www.nationalccdc.org/

[12] Ihaka, R. and R. Gentleman (1996). "R: A language for data analysis and graphics," Journal of Computational and Graphical Statistics , 5 , 299-314.

[13] Bronson, G. (2010). C++ For Scientists and Engineers, 3rd edition, Course Technology/CENGAGE Learning.

[14] Zak, D. (2013) Programming with Microsoft Visual Basic 2012, Course Technology/CENGAGE Learning.

[15] Seitz, J. "Black Hat Python: Python Programming for Hackers and Pentesters", ISBN 978-1-59327-590-7, No Starch Press, 2015.

[16] Skull Security Hacked Password Repository Website: https://wiki.skullsecurity.org/Passwords

[17] Grace Hopper 2018 Conference @ Anita B.ORG: https://ghc.anitab.org/ghc-18/

[18] Zodiac Killer Ciphers: http://www.zodiackillerciphers.com/

[19] ASCII code: https://en.wikipedia.org/wiki/ASCII

[20] UNICODE: https://en.wikipedia.org/wiki/Unicode

[21] UTF-8: https://en.wikipedia.org/wiki/UTF-8

[22] The Simon Singh Black Chamber (www.simonsingh.net/The_Black_Chamber/chamberguide.html).