University of Louisville
ThinkIR: The University of Louisville's Institutional Repository

Electronic Theses and Dissertations

7-2009

Secure messaging with Apple's iPhone.

Todd Randal Grooms
University of Louisville

Follow this and additional works at: https://ir.library.louisville.edu/etd

Recommended Citation
Grooms, Todd Randal, "Secure messaging with Apple's iPhone." (2009). *Electronic Theses and Dissertations.* Paper 537.
https://doi.org/10.18297/etd/537

This Master's Thesis is brought to you for free and open access by ThinkIR: The University of Louisville's Institutional Repository. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of ThinkIR: The University of Louisville's Institutional Repository. This title appears here courtesy of the author, who has retained all other copyrights. For more information, please contact thinkir@louisville.edu.

SECURE MESSAGING WITH APPLE'S IPHONE

By

Todd Randal Grooms
Bachelor of Science in Computer Engineering and Computer Science
J.B. Speed School of Engineering, University of Louisville, 2008

A Thesis
Submitted to the Faculty of the
University of Louisville
J.B. Speed School of Engineering
as Partial Fulfillment of the Requirements
for the Professional Degree

MASTER OF ENGINEERING

Department of Computer Engineering and Computer Science
J.B. Speed School of Engineering, University of Louisville
Dr. Ibrahim N. Imam, Thesis Advisor

July 2009

SECURE MESSAGING WITH APPLE'S IPHONE

By

Todd Randal Grooms

Presented

July 23, 2009

Read and approved by the examination committee

____

Dr. Ibrahim N. Imam, Thesis Director

____

Dr. Ahmed Desoky

____

Dr. Julius P. Wong

DEDICATION

This thesis is dedicated to my grandfather.

His last wish was to see me graduate from college;
Unfortunately, he did not get to see it in this life.
I love you pappaw.

// ACKNOWLEDGMENTS

There are many people who are directly and indirectly responsible for where I am today. Without their help, I would have never accomplished what I have.

First, I would like to recognize my family. Although they have their quirks, I would not have them any other way. I love all of them and will always be grateful for their support throughout my life. A few that stand out are my brother Chris, my father John, and my grandmother Wanda. All three of these people listened to their fair share of my complaints and always lent an ear when needed. Their guidance throughout my life has been invaluable and I love them dearly.

I also want to recognize many of the teachers that I have had throughout my academic career. I want to thank Jon Summerville for encouraging me not to settle and that I can always do better. I want to thank all of the teachers during my time at Mayfield High School as well. They were wonderful and always supportive, especially Mrs. Miller who instilled a love for calculus that helped get me through the rough path ahead.

Next, I would like to recognize the faculty of Speed School for their guidance and support. They were very helpful and enabled me to learn during my time at the University of Louisville.

Last, but not least, I would like to recognize my lovely fiancé, Ashley. She has kept me sane throughout college and without her I would not be the person I am today.

ABSTRACT

Electronic communication is gradually becoming a larger part of today's society. This type of communication not only encompasses ways people stay in touch, but also how people manage or share their personal data. As this type of communication increases, the protecting of sensitive data becomes a goal amongst users. The goal of this thesis is to offer a secure solution for mobile devices to share data with other mobile devices.

The idea is simple: provide a software solution that can be ported to multiple mobile device platforms that will allow a user to securely encrypt a message that can be sent using existing communication technologies from one mobile device to another, where that encrypted message can then be decrypted and used by the recipient. The method should rely on a common passkey that the sender and recipient agree upon through verbal communication in advance that can be used to encrypt and decrypt messages that are transmitted between them. The passkey should also never be documented (in the application or elsewhere) to prevent it from being discovered by an attacker.

Although it would be possible to write this application on different platforms, the result is an application that is used on Apple's iPhone. The application uses the secure open-source Blowfish cipher. A sender can type in their plain-text message and a passkey. The application can then encrypt the

plain-text message and output a cipher-text message that the sender can then copy and paste into a text message or email. The recipient can copy the cipher-text into the application's message field along with the shared passkey and then decrypt the message into its original plain-text format. This method offers the best flexibility, by taking advantage of technologies that mobile device customers use frequently, and a solid secure method of protecting their messages by way of the Blowfish cipher algorithm.

The application presented in this thesis offers a way to enable secure messaging with mobile devices. With that being said, this application is not meant to be the final answer to this problem, especially since the current iteration has not been ported to other platforms. The application could also be modified to offer better support in the transmission of these messages as well to offer a better experience to the end user. With this introductory application, it is hoped that security in digital communications will be brought under the speculation of its users so that more secure options will become available.

TABLE OF CONTENTS

DEDICATION		**iii**
ACKNOWLEDGMENTS		**iv**
ABSTRACT		**vi**
TABLE OF CONTENTS		**vii**
LIST OF PSEUDOCODE		**viii**
LIST OF FIGURES		**ix**
1	**INTRODUCTION**	**1**
	1.1 SMS: How Secure is It?	2
	1.2 You Have Email Though, Right?	4
	1.3 The Proposed Solution	4
	1.4 Thesis Objectives	6
	1.5 Thesis Structure	7
	1.6 Application Assumptions	9
2	**TECHNOLOGICAL INGREDIENTS**	**10**
3	**DESIGN AND IMPLEMENTATION**	**15**
	3.1 Graphical User Interface	15
	3.2 Blowfish Cipher	23
	3.3 Data Conversion	25
4	**TESTING AND DIFFICULTIES**	**31**
	4.1 Cipher Text Output Issues	31
	4.2 Unexpected GUI Actions	33
5	**CLOSING AND FUTURE CONSIDERATIONS**	**35**
APPENDIX A		**39**
APPENDIX B		**40**
APPENDIX C		**48**
APPENDIX D		**49**
LIST OF REFERENCES		**56**

LIST OF PSEUDOCODE

3.1	The Blowfish Encryption Algorithm	24
3.2	The Blowfish F Function	24
3.3	Input Message Manipulation for Encryption Method	27
3.4	Input Message Manipulation for Decryption Method	30

LIST OF FIGURES

2.1	iPhone Simulator (Home Screen)	10
2.2	iPhone Simulator (Application)	10
2.3	Xcode	11
2.4	Interface Builder	12
3.1	The Main Interface (Encrypt)	15
3.2	Ready to Enter a Message (Encrypt)	16
3.3	Password Entered (Encrypt)	17
3.4	Copy Encrypted Message	18
3.5	The Main Interface (Decrypt)	19
3.6	Ready to Enter a Message (Decrypt)	19
3.7	Password Entered (Decrypt)	20
3.8	Decrypted Message	21
3.9	Blank Message Error (Encrypt)	21
3.10	Blank Message Error (Decrypt)	21
3.11	Maximum Password Length Error	22
3.12	Minimum Password Length Warning	23
3.13	The Blowfish Diagram (via Wikipedia)	25

I. INTRODUCTION

In today's world, people are connected more than ever before. With the evolution of the computer and mobile phone, people are always connected. We are able to receive order updates via email and bank notifications via Short Message Service (SMS) messaging. People are more concerned with staying connected more than ever before. With that type of transparency though, privacy questions begin to cloud the consciousness of our minds. Personal users are not the only ones concerned with keeping their data private; an increasing number of companies are looking for ways to keep their data secure, especially as employees are able to access sensitive company information from their mobile devices while traveling.

One of the more popular types of devices for keeping connected while traveling is the cellular phone. Cellular phones have evolved into a necessity as they have transformed from a device that is able to maintain voice communication from remote locations across the world to devices that can receive email, access the internet, and perform large calculations in a timely manner.

The largest growth field for cellular phone manufacturers is the smartphone field. A smartphone is a phone that has a powerful operating system that allows expandability, the ability to connect to the internet for browsing and email, and

has a built-in keyboard; a smartphone is essentially a computer for your pocket. According to the Wall Street Journal[1], first-quarter 2009 sales of cellular handsets was down 9.4% from a year ago. In contrast, they report that smartphone sales increased 11% from last year. These numbers indicate that people are starting to buy a smartphone instead of a regular handset. As more people invest in a smartphone, they start to use the capabilities of the handset to access private data whether it be job related or for personal use.

While it is very convenient to access sensitive data quickly from anywhere, there are certain risks involved in doing so. The two most popular ways of receiving information from handsets are from SMS messaging and email. While each one can be relatively safe, neither can offer complete protection from unwanted eyes.

1. SMS: How Secure is It?

SMS messages are designed to be sent directly to the recipient's device. Once a user types a message, enters the recipient's phone number, and hits send, that message is sent straight to the recipient's handset. Cellular companies only keep a copy of that message on their servers for a limited amount of time (to verify the message has been sent). After that, the message is removed from their servers. The length of time the message resides on the server varies on the provider, but that length of time is always less than a month (Wendland 2008).

So if there are no long term records of the messages, they must be pretty safe, right? Not exactly. A recent case (November 2002) involved network

[1] The statistic was obtained by the Wall Street Journal from Gartner, Inc.

engineers for the British wireless carrier, O2, intercepting SMS message contents and showing the contents of those messages to an unauthorized user (Leyden 2002). From this story in *The Register*, Gartner, a well known technology consulting firm, was quoted as saying, “The contents of SMS messages are known to the network operator’s systems and personnel. Therefore, SMS is not an appropriate technology for secure communications. Most users do not realise how easy it may be to intercept.” Of course this case is not necessarily a lack of security on the technology side, but rather a lack of security in personnel for providers. However, this could still lead to sensitive information being breached by unauthorized users.

Another security risk with SMS messaging is the physical risk. If a user leaves their phone in a public place, the device could fall into the wrong hands. Although many handsets come with some sort of pass-key or lock (a software lock that requires a certain key-press combination to unlock the device), these features are not enabled by default. If a user does not enable these, someone can easily look at their SMS messages. Once a malicious user has the device and is able to access the handset’s SMS message inbox, there is not another security layer to prevent them from reading those received messages. Not only does the unauthorized user have access to the received messages, they now have access to the messages that have been sent from the device. Handset manufacturers have been improving their “hands on” security (protecting handsets from malicious users that can physically access the handset), but these

security features are rarely enabled by default, which leaves users open to an attack.

2. You Have Email Though, Right?

Due to smartphone support for email, this method is for the most part secure. As long as the email provider has a secure setup that uses Secure Sockets Layer (SSL), or its successor Transport Layer Security (TLS), for retrieving or sending an email, a user is well protected from packet sniffing during transmission. The only security risks that could arise are that of a weak email account password or, again, physical access to the handset. As weak passwords are a risk to any system, this scenario will be overlooked. However, physical access to the handset can be a problem. Many handsets, when setting up email accounts, save the email password so that the mail can be downloaded to the device as new mail is received; if a user does not have some sort of pass-key/lock on their device, their email could be read by an unauthorized user.

3. The Proposed Solution

Secure messaging on handheld devices should take advantage of technology already available, but also incorporate new technology as well. The solution proposed is to create a program capable of encrypting and decrypting a message that sits on the handheld device. For transmission of the message, technologies already available should be used (such as SMS messaging or sending the message in an email).

The sender opens up the encryption/decryption application on their handheld device. From there they type in their message, select to encrypt the

message, and enter a password to encrypt the message with. After this, the encrypted message is generated and displayed to the sender. The sender then copies the encrypted text into a message to be sent to the receiver. At this time, the sender needs to give the receiver the password so that the message can be decrypted. Once the receiver receives the message, they copy the encrypted message into the encryption/decryption application on their handheld device. They select to decrypt the message and enter the password given to them by the sender. The decrypted message is then displayed to the receiver.

The encryption and decryption steps need to take place on the handheld devices for maximum security and optimum compatibility with current technologies. If the message is not encrypted on the device before transmission, then phone manufacturers would have to change the transmission technology (either SMS or email) to perform the encryption or mobile carriers and email providers would have to encrypt the messages on the server. It would be difficult to change technologies that are currently in place, such as SMS messaging. If encryption during transmission of the SMS message was desired, then all SMS messaging clients would have to be updated to do this. This would be an unrealistic solution as this standard has already been set. If the message was encrypted on a server, then not only is an unencrypted message sent to the server, but also the password must accompany that message to be encrypted later. This would allow an unauthenticated user to not only hijack the message, but to also obtain the password to use at a later time. However, if the encryption and decryption processes take place on the device before transmission, it would

require no change in current technologies (SMS or email) and would ensure safe transmission.

Another security feature in this method is that the password is not stored anywhere. Once the receiver gets the encrypted message, they use the password given to them by the sender to decrypt the message. If the receiver types in the wrong password, a message is still displayed (just the wrong one). Since the password is not stored, it is impossible to check if the receiver has input the correct password and warn them. If someone were trying to get the password, this would help during an automated, brute-force attack; the attacker would have to read each decrypted message to see if it was the right one.

4. Thesis Objectives

There are a few objectives that this thesis is meant to accomplish. Those objectives are as follows:

- Learn to develop an application for the iPhone
- Create an application that acts as a layer between creating a message and sending a message; The purpose of this layer is to encrypt and decrypt messages using the Blowfish cipher
- Successfully encrypt and decrypt messages within the application.

Prior development experience will be relied upon to complete these objectives. While a developer may not have first hand knowledge of a specific platform in which they are to develop an application for, it is possible to use past computer science experience to accomplish this task. While iPhone development uses a programming language, Objective-C, that is foreign to some users, they may not

be exactly unfamiliar with it; Objective-C shares common syntax with the C language and is heavily based on it. Therefore C programming experience will be utilized during development. Another potential hurdle will be learning the Cocoa Touch application programming interface (API). This abstraction layer is a way for iPhone developers to use system resources built into the iPhone operating system (OS) without a ton of extra work.

It is also important to note that the iPhone has somewhat limited hardware resources. The model iPhone that will be used for this development (iPhone 3G) has 16GB of storage and only 128MB of RAM. Previous development experience on micro controllers (such as the 68HC11) will become useful for completing this thesis as it will be important to ensure that application is efficient and only uses the system resources that it needs to use.

5. Thesis Structure

This thesis is not meant to be the final answer when it comes to secure messaging from handheld devices. As a matter of fact, many devices on the market today still lack the power or tools to make this idea into a reality. However, this demonstration of secure messaging could become a reality in the future as all mobile phone devices start to become more computer like and include developer tools. Once all phones include developer tools, they could be equipped with an encryption/decryption program to make this idea of a secure messaging platform for mobile devices into a reality. This thesis is also not meant to show why secure messaging for mobile devices is useful, but rather how it could be achieved on a small scale.

The next chapter discusses the software used to make this application possible. It will include a summary of the programming language and information about the programming environment that was used. That chapter will also discuss the background of the cipher algorithm used and why that particular algorithm was chosen.

The third chapter will break down some of the application. Important parts of the code will be examined and explained, focusing on how the cipher algorithm is implemented and especially looking at its inner workings. It will be important to know exactly how the plaintext is manipulated and processed through the encryption algorithm, how it is displayed when in its encrypted form, and then how it is processed through the decryption algorithm. This chapter will also examine the graphical user interface (GUI) that will be used for the application.

The fourth chapter will discuss the testing involved with the application. Some test cases were needed to make sure that the application was functioning properly and efficiently; this chapter will discuss how these test cases were created and used. This chapter will also cover some of the major issues encountered and how they were resolved.

The fifth and final chapter will discuss possible future enhancements. Although the idea of secure messaging to and from handheld devices is important, this chapter will look at what needs to be done to make this application ideal for such a purpose.

6. Application Assumptions

The only assumption for this application is that only text is given as an input for encryption or decryption. This means that this iteration does not currently work with pictures, audio files, or video files yet. While it is possible to write encryption schemes for an entire file, that is not the purpose of this application.

II. TECHNOLOGICAL INGREDIENTS

The hardware platform used for this project was chosen because of its availability and growing popularity with developers of all backgrounds. The mobile device used is Apple's iPhone. Apple released the iPhone in the United States on June 29, 2007. The first version of the iPhone did not support third party applications, but this changed with the release of the second version if the iPhone and the second version of the iPhone operating system on July 11, 2008. With the iPhone OS 2.0, developers were given the iPhone software development kit (SDK) for free to start developing third party applications. The free membership to Apple's developers program included the ability to test software on an iPhone simulator (part of the SDK).

FIGURE 2.1 - iPhone Simulator (Home Screen)

FIGURE 2.2 - iPhone Simulator (Application)

If the developer wishes to test the software out on actual hardware (via an iPhone or iPod Touch), the developer has to join the developers program with a paid membership.

The iPhone SDK was an addition to Apple's integrated development environment (IDE), Xcode. Xcode is a free IDE that is distributed by Apple. This IDE allows for development on iPhone/iPod Touch applications, Mac applications, and Safari web applications. Xcode contains all the tools to organize a project, edit code (with an extremely powerful text editor with predictive code), compile the code, debug the code, and run the application.

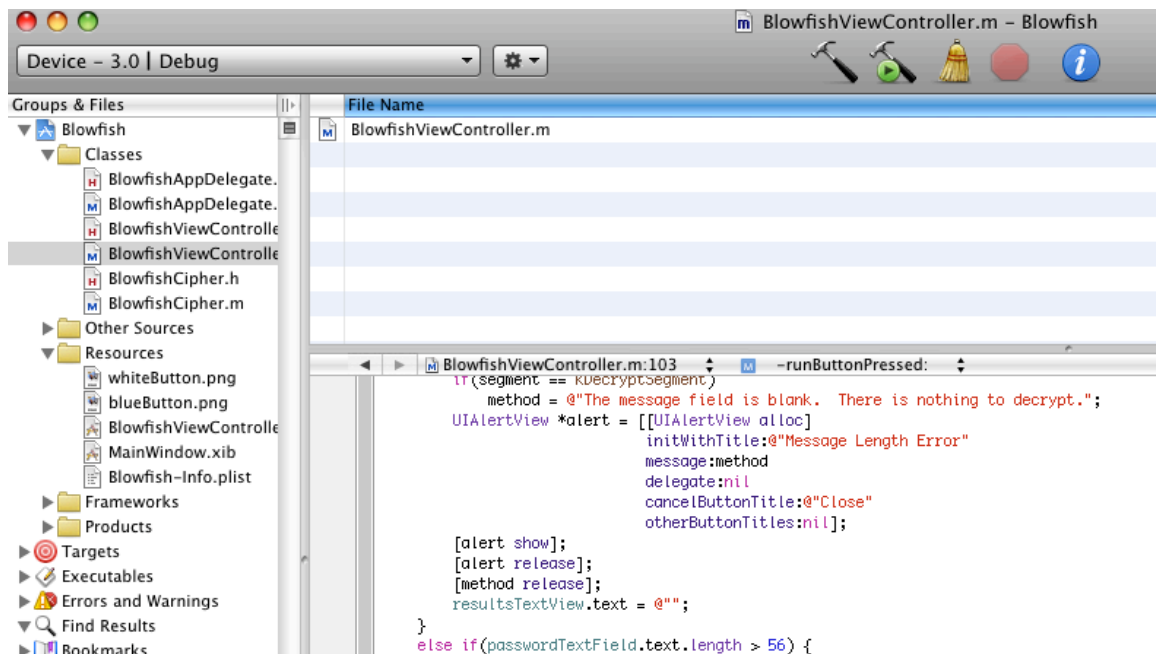

FIGURE 2.3 - Xcode

To build the interface, Apple includes an application called Interface Builder. Interface Builder is a powerful graphical user interface (GUI) creation tool that is included as a part of Apple's developer tools.

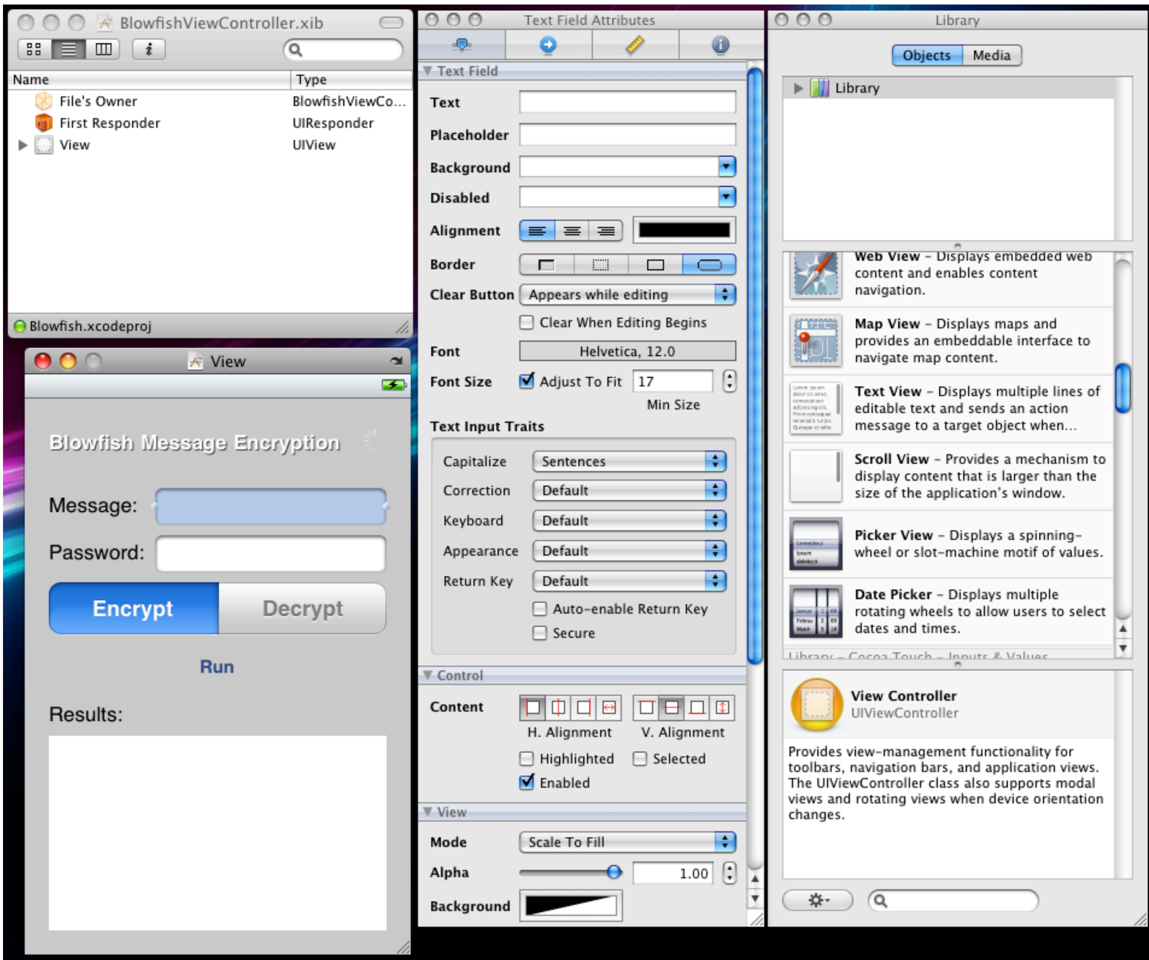

FIGURE 2.4 - Interface Builder

As of this writing, iPhone development is only capable on Apple computers running Mac OS X.

To perform the encryption and decryption of messages, the Blowfish cipher, a variable length key, 64-bit block cipher, was chosen because of its open-source license, strength, and familiarity. The author of the Blowfish cipher, Bruce Schneier, states on his website that, “Everyone is welcome to download Blowfish and use it in their application.” The algorithm is not patented and, according to the author, will remain so. He has placed it in the public domain, so that anyone can use it.

Even though the cipher is well documented online, there is no known cryptanalysis of the cipher as of 2009. According to Schneier:

> Many cryptographers have examined Blowfish, although there are few published results. Serge Vaudenay examined weak keys in Blowfish; there is a class of keys that can be detected–although not broken–in Blowfish variants of 14 rounds or less. Vincent Rijmen's Ph.D. thesis includes a second-order differential attack on 4-round Blowfish that cannot be extended to more rounds.

To implement this cipher, the algorithm was ported from a previous school project where it was written in the Java programming language.

Applications for the iPhone or iPod Touch hardware are developed in the Cocoa Touch application programming interface (API), which was developed by Apple. Cocoa Touch is based on the Cocoa API which is used for development on the Mac OS X operating system. Cocoa is based on the Model-View-Controller (MVC) principles.

The primary language for Cocoa is Objective-C. Objective-C was originally made popular by NeXT, Inc. as the main language on their operating system, NeXTSTEP. Although Objective-C is heavily associated with Apple and their platform, Objective-C programs that do not depend on Cocoa can be developed on any system that is supported by the GNU Compiler Collection (GCC). Brad Cox and Tom Love are the main creators behind Objective-C. The language is intended to be an object-oriented evolution of the C language. The Smalltalk programming language was a heavy influence on the syntax in Objective-C.

For message delivery, it was decided that this application should use current technologies already in existence. The two most popular forms of sending text data on mobile devices are through SMS messaging or email. Although SMS messaging would be a better fit (device-to-device messaging, no long term

storage of messages), it may be necessary to use email. One instance in particular where email would be a better fit is in the case of using this application on an iPod Touch. An iPod Touch is like an iPhone sans phone, which rules out the possibility of using SMS messaging on this device. With this in mind, this application is designed with that compatibility in mind, therefore SMS messaging and email can be used with this application.

III. DESIGN AND IMPLEMENTATION

This chapter will focus on three aspects of the application: the graphical user interface (GUI), the blowfish cipher algorithm, and the methods involved with converting a string input into something usable by the cipher. Where necessary, pseudocode will be used to help explain algorithms; a full listing of working code can be found in the appendices.

1. Graphical User Interface

The interface for this application was designed with ease of use and functionality in mind. The layout is clean and self explanatory.

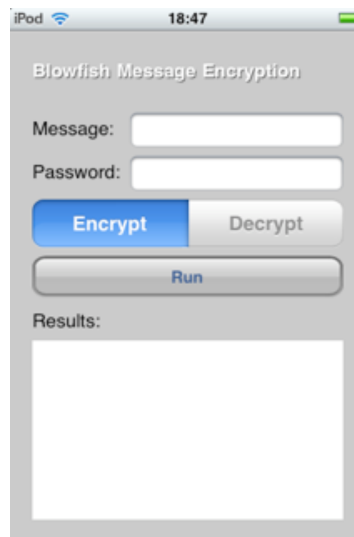

FIGURE 3.1 - The Main Interface (Encrypt)

There are two text-fields towards the top of the application, one is for the message and the other is for the password, both of which are appropriately

named. Below that lies a selection switch that allows the user to select whether they intend to encrypt the message or decrypt the message. The selection switch is followed by a run button that starts the encryption or decryption process. Finally, the bottom half of the screen is a text-area in which the result is displayed to the user so that they can view it or copy it.

When the user clicks on the “Message” text-field, a keyboard is displayed and the user can type in their message, or if they tap again, they can paste an already copied message, into the field. After they are finished, they can tap the “return” key on the bottom, right-hand part of the screen to jump to the “Password” text-field, or they can move focus to the “Password” text-field by tapping in the text-field.

FIGURE 3.2 - Ready to Enter a Message (Encrypt)

The “Password” text-field works much like the text-field before it, but with a few minor differences. One of those differences, which will be noticed instantly upon typing, is that the last character typed will be shown briefly, then masked

with a black circle so that wondering eyes will not be able to read the entire password. While the password is eventually masked, it is possible for the password to be seen by someone if they watch the user enter the entire password (since the characters are echoed onto the screen for a brief time). Therefore, it is still important to be cautious when entering this password and the user should not let someone see the screen while they type in their password. When the user is finished typing in their password, they can close the keyboard by tapping the "Done" key on the bottom, right-hand part of the screen.

FIGURE 3.3 - Password Entered (Encrypt)

Next the user taps the "Run" button on the middle of the screen to encrypt the message. The result is a hexadecimal string that is displayed in the "Result" text-area. If the result is larger than the text-area, the user can scroll vertically within the area to view all its contents. Once the user is ready to transmit the message, they can tap twice on the string which will bring up a "Copy" dialog. This will allow the user to copy the encrypted message so that they can paste it

into an SMS message or into an email message and send it to whomever they wish. While this method is not very streamlined, it does offer the greatest flexibility for the end user. It is also important to note that the password should not be shared within the SMS message or email message as this would harm the security of this message; if possible, the password should be shared from the sender to the recipient secretly, with only them knowing the password.

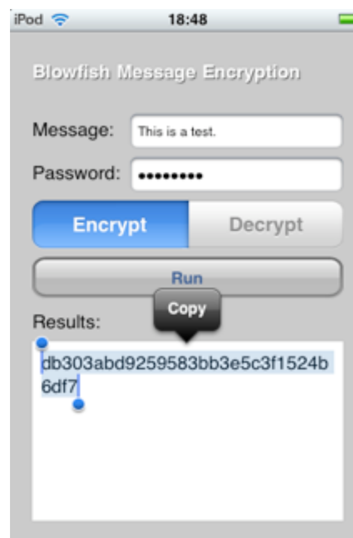

FIGURE 3.4 - Copy Encrypted Message

The workflow for decrypting a message is very similar to that of encrypting a message, but with a few differences. First the message that needs to be decrypted needs to be copied onto the clipboard. The message can be entered by hand, but this could become cumbersome to the user; therefore, it is preferred to have already copied the message. Next, the user opens up the application. Now the user needs to make sure the application is set for decryption. This is done by switching the selection switch to the “Decrypt” position. This can be done at the beginning or at anytime before the user presses the “Run” button.

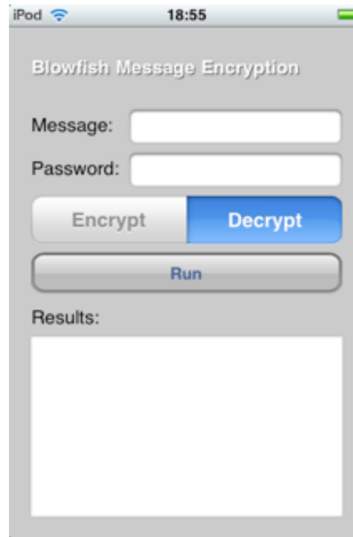

FIGURE 3.5 - The Main Interface (Decrypt)

Next the user taps the “Message” text-field. If the cipher text has been copied to the clipboard, the user can tap the “Message” text-field again to bring up the “Paste” menu to paste the cipher text into the field; otherwise, the user would have to type in the cipher text by hand. After the cipher text has been entered, the user can tap the “return” key to move down to the “Password” text-field.

FIGURE 3.6 - Ready to Enter a Message (Decrypt)

The next step is for the user to type in the password. The password entered for decryption has to be the exact same password that was used for encryption. When the password has been entered, the user can tap the “Done” key to collapse the keyboard.

FIGURE 3.7 - Password Entered (Decrypt)

Finally the user can tap the “Run” button which will decrypt the cipher text and display the result in the “Result” text-area. If the resulting plain text is larger than the text-area, the user can scroll vertically to read the entire message. If desired, this message can be copied as well, as the copy/paste functionality is a part of the phone’s operating system and is available to all applications.

FIGURE 3.8 - Decrypted Message

The program also performs some error checking on a user's input. If the message field is left blank, the user will be presented with one of two messages, depending on whether the user was attempting to encrypt or decrypt.

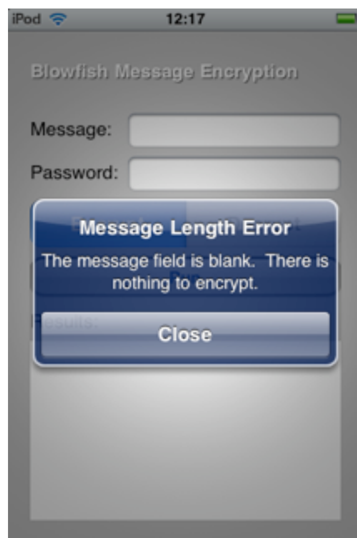

FIGURE 3.9 - Blank Message Error (Encrypt)

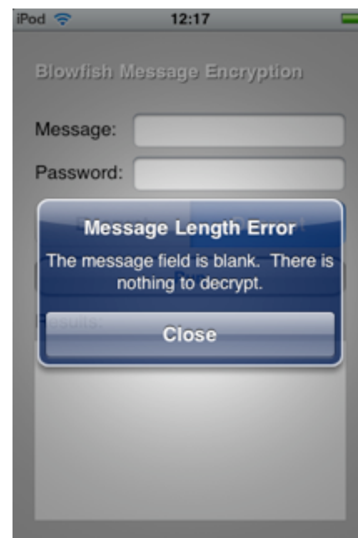

FIGURE 3.10 - Blank Message Error (Decrypt)

The user is notified of their error and the program does not run the encryption or decryption process as there is no message to perform the task on. The only other error that can be encountered with this application is when the user enters

a password that is too long. The Blowfish cipher is designed to work with a variable key length between 32 bits and 448 bits. Every character is equal to eight bits. Therefore, the cipher needs a password with a length of four up to 56.

FIGURE 3.11 - Maximum Password Length Error

Since the algorithm deals with 32 bit chunks, it is preferred to run the algorithm with a key (password) of at least four characters in length (32 bits) but the algorithm will safely run with less, but it will be a very weak encryption. Therefore, this program will still run with passwords less than four characters in length, but the user will be presented with a warning that cautions against this practice.

FIGURE 3.12 - Minimum Password Length Warning

2. Blowfish Cipher

The Blowfish cipher is well documented by its author, Bruce Schneier, on his website. He has included multiple implementations of the algorithm in many different languages. Instead of looking at the specifics of this algorithm in code, which can be found on the author's website, it is more beneficial to look at what the algorithm is doing in theory.

The process starts by taking the P array, length of 18, and the four S-Boxes, length of 256 each, and initializing them with the hexadecimal digits of PI (excluding the first three). Next the password, or key, is taken and the algorithm performs the XOR operation with it and the P array. An XOR operation is performed with the first element of the P array and the first 32 bits of the key, then another XOR operation is performed with the second element of the P array and the second 32 bits of the key. This continues for all the elements of the P array, repeating the key bits if necessary. A key is not to exceed the fourteenth

element of the P array, meaning there is a maximum key of 56 characters (448 bits).

The next step is to encrypt an all-zero string with the Blowfish algorithm. The result of this replaces the first and second element of the P array. That same result is then encrypted again (with the modified P array) and that result replaces the third and fourth element of the P array. This continues until all elements of the P array and the S-Boxes (in order) have been modified, thus giving pass key dependent sub-keys (P array and S-Boxes).

The encryption method pseudocode, as given by Schneier, starts with a single input (x in this case):

```
Divide x into two 32-bit halves: xL, xR
For i = 0 to 15:
xL = xL XOR P[i]
xR = F(xL) XOR xR
Swap xL and xR
Next i
Swap xL and xR (Undo the last swap.)
xR = xR XOR P[16]
xL = xL XOR P[17]
Recombine xL and xR
```

PSEUDOCODE 3.1 - The Blowfish Encryption Algorithm

The F function that is given in the pseudocode above, is given as

```
Divide xL into four eight bit quarters: a, b, c, and d
F(xL)=((S-Box_1[a]+S-Box_2[b] % 2^32) XOR S-Box_3[c])+S-Box_4[d] % 2^32
```

PSEUDOCODE 3.2 - The Blowfish F Function

These pseudocode listings assume that the P array and the S-Boxes are given as array elements in the code.

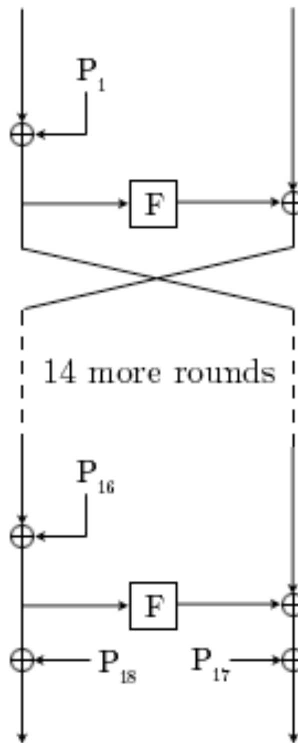

FIGURE 3.13 - The Blowfish Diagram (via Wikipedia)

The decryption method is exactly like the encryption method, except that the P array is used in reverse order. The for loop counts down from the eighteenth element in the P array through the third element in the P array. Then an XOR is performed on the second element and the variable xR, and another XOR is performed on the first element and the variable xL. Finally the xL and xR variables are recombined to finish the method.

3. Data Conversion

The first thing that happens when the user taps the “Run” button, is the message text field and password text field are saved. These text fields have the input stored in a string data type, but a conversion is formed to convert these

strings into a character array. The password character array is used to initialize the Blowfish instance. The message character array takes a different path depending on whether the user is performing an encrypt function or a decrypt function.

If the user is performing an encrypt function, the process is straightforward. Since the Blowfish encryption algorithm takes a 64-bit input (the 64-bit input is halved to make two 32-bit legs as explained earlier), the input characters need to be combined into a suitable input for the algorithm. This implementation of the Blowfish encryption algorithm expects the 64-bit input to be halved before the algorithm is called. Therefore, two 32-bit hexadecimal starting variables are created (xL and xR, initially equal to 0x00000000).

The first four characters go onto the xL variable and the next four characters go onto the xR variable. To "push" these characters onto these variables, a method of performing a bitwise OR operation on a character and the variable, and then performing a bitwise left shift on the variable is used. This method is performed repeatedly until the first eight characters have been used. Then the xL and xR variables are sent through the Blowfish encryption method. Once the first eight characters have been encrypted, they are appended to a results string in hexadecimal form, with care to ensure that leading zeros are attached as well. Since numbers that are printed out typically have their leading zeros omitted, every eight bits of hexadecimal output is taken and compared to 0x10; if the output is less than 0x10, a leading zero is appended before the eight bit hexadecimal is appended. This process continues until all of the characters in

the input character array have been encrypted. Because this process takes eight characters at a time, if the number of input characters modulus eight is greater than zero, the ASCII character 0x00 is appended to the end before encryption. Since this character is the equivalent of NULL, when the encrypted string is decrypted later.

```
String result
for(iterator = 0; iterator < message_length; iterator += 8) {
   unsigned long xL = 0x00000000, xR = 0x00000000
   for(counter = 0; counter < 3; counter++) {
      if(iterator + counter < message_length)
         xL = xL OR message[iterator + counter]
      Shift xL 8-bits to the Left
   }
   if(iterator + 3 < message_length)
      xL = xL OR message[iterator + 3]

   for(counter = 4; counter < 7; counter++) {
      if(iterator + counter < message_length)
         xR = xR OR message[iterator + counter]
      Shift xL 8-bits to the Left
   }
   if(iterator + 7 < message_length)
      xR = xR OR message[iterator + 7]

   [Blowfish_Encrypt:&xL and:&xR]

   unsigned long mask_1 = 0xFF000000, mask_2 = 0x00FF0000
   unsigned long mask_3 = 0x0000FF00, mask_4 = 0x000000FF
   unsigned long temp;

   temp = xL AND mask_1
   Shift temp 24-bits to the Right
   if(temp < 0x10)
      [result append:"0"]
   [result append:temp_in_char_form]
   temp = xL AND mask_2
   Shift temp 16-bits to the Right
   if(temp < 0x10)
      [result append:"0"]
   [result append:temp_in_char_form]
   temp = xL AND mask_3
   Shift temp 8-bits to the Right
   if(temp < 0x10)
      [result append:"0"]
   [result append:temp_in_char_form]
   temp = xL AND mask_4
   if(temp < 0x10)
```

```
      [result append:"0"]
   [result append:temp_in_char_form]

   temp = xR AND mask_1
   Shift temp 24-bits to the Right
   if(temp < 0x10)
      [result append:"0"]
   [result append:temp_in_char_form]
   temp = xR AND mask_2
   Shift temp 16-bits to the Right
   if(temp < 0x10)
      [result append:"0"]
   [result append:temp_in_char_form]
   temp = xR AND mask_3
   Shift temp 8-bits to the Right
   [result append:temp_in_char_form]
   if(temp < 0x10)
      [result append:"0"]
   temp = xR AND mask_4
   if(temp < 0x10)
      [result append:"0"]
   [result append:temp_in_char_form]
}
display result;
```

PSEUDOCODE 3.3 - Input Message Manipulation for Encryption Method

The decryption function is a little trickier, but still follows the same basic guidelines as the encryption function. The big difference is in how the message character array is handled. Since the application prints out the encrypted text as a hexadecimal string, the input message for decryption is in hexadecimal format already. Instead of reading in the input character array and manipulating it via the char data type to ASCII hexadecimal value conversion that was used before, a special conversion needs to be done.

The input character array is already in hexadecimal format, so a switch/case statement is used to match up a character to its hexadecimal value (converting a hexadecimal character to its unsigned long value). An example of this would be if the input character was ‘A’ then the unsigned long value that would be returned

would be 0x0000000A. After this conversion is done, these hexadecimal values can be combined much like they were before.

At the end of this function, when printing out the decrypted message, there are steps that need to be taken to ensure the message prints out as expected. At first the decrypted message is an unsigned long hexadecimal value. However, each eight bits is actually an ASCII hexadecimal value for the characters that will need to be displayed to the user. To ensure each character is displayed correctly to the user, the unsigned long hexadecimal values have to be broken apart into eight bit chunks. These eight bit chunks are then appended to the string that will be displayed for the output by forcing each eight bit hexadecimal value to be displayed as a character. When displaying these eight bit hexadecimal values as characters, Objective-C converts the ASCII hexadecimal value to the character value.

For example, the 64-bit hexadecimal value that is given after being decrypted could be 0x48656C6C6F000000. This value can be broken down into eight bit sections: 0x48, 0x65, 0x6C, 0x6C, 0x6F, 0x00, 0x00, 0x00. When converting these ASCII hexadecimal values to characters, the output would be: ‘H’, ‘e’, ‘l’, ‘l’, ‘o’ (the 0x00 ASCII hexadecimal values are equal to the NULL character and therefore not displayed). This process ensures that the hexadecimal value outputs given by the decryption function are converted into a readable output.

```
String result
for(iterator = 0; iterator < message_length; iterator += 16) {
   unsigned long xL = 0x00000000, xR = 0x00000000
   for(counter = 0; counter < 7; counter++) {
      if(iterator + counter < message_length)
         xL = xL | [hex_Char_To_Long:message[iterator + counter]]
      Shift xL 4-bits to the Left
   }
   if(iterator + 7 < message_length)
      xL = xL | [hex_Char_To_Long:message[iterator + 7]]

   for(counter = 8; counter < 15; counter++) {
      if(iterator + counter < message_length)
         xR = xR | [hex_Char_To_Long:message[iterator + counter]]
      Shift xL 4-bits to the Left
   }
   if(iterator + 15 < message_length)
      xR = xR | [hex_Char_To_Long:message[iterator + 15]]

   [Blowfish_Decrypt:&xL and:&xR];

   unsigned long mask_1 = 0xFF000000, mask_2 = 0x00FF0000
   unsigned long mask_3 = 0x0000FF00, mask_4 = 0x000000FF
   unsigned long temp;

   temp = xL AND mask_1
   Shift temp 24-bits to the Right
   [result append:temp_in_char_form]
   temp = xL AND mask_2
   Shift temp 16-bits to the Right
   [result append:temp_in_char_form]
   temp = xL AND mask_3
   Shift temp 8-bits to the Right
   [result append:temp_in_char_form]
   temp = xL AND mask_4
   [result append:temp_in_char_form]

   temp = xR AND mask_1
   Shift temp 24-bits to the Right
   [result append:temp_in_char_form]
   temp = xR AND mask_2
   Shift temp 16-bits to the Right
   [result append:temp_in_char_form]
   temp = xR AND mask_3
   Shift temp 8-bits to the Right
   [result append:temp_in_char_form]
   temp = xR AND mask_4
   [result append:temp_in_char_form]
}
display result;
```

PSEUDOCODE 3.4 - Input Message Manipulation for Decryption Method

IV. TESTING AND DIFFICULTIES

Testing this application, as well as other encryption applications, takes patience and time. Due to the nature of the process, the user does not know if the application works as expected until they encrypt a message and safely decrypt the cipher text and ensure that the result is accurate. Therefore, testing consisted of a manual approach, repeatedly attempting to break the application so that the error could be found and fixed.

This chapter will cover only the larger problems and their solutions. Many errors occurred, but the majority of them were syntax errors. The bugs that were left that needed to be solved were cipher text output issues and unexpected graphical user interface (GUI) actions.

1. Cipher Text Output Issues

With a correct encryption/decryption program, the input plain text and the output plain text always match up. For the most part, most of the input plain text was matching the output plain text. However, in what became the strangest bug of the entire project, occasionally the input plain text did not match the output plain text. In these cases, the output plain text would be gibberish, or what seemed to be completely random characters. This can be one of the scariest types of bugs, as it can sometimes be hard to determine what is causing this unexpected behavior.

Luckily, the problem was quickly noticed and the reason behind the problem was apparent. The problem was occurring during the encryption process. The Blowfish cipher works with 64-bits of the input at a time (or in this application's instance, with eight characters at a time). During this process, if the input is less than eight characters, the input is padded at the end with 0x00 characters (the null character). If all goes well during the encryption process, the resulting hexadecimal cipher text will have a character length that is a multiple of eight (i.e. the cipher text length will be eight characters in length, sixteen characters in length, twenty-four, etc).

However, in the first iteration of the application, a four character string was input through the encryption algorithm and the resulting cipher text was less than eight characters in length. This problem was caused by not printing out the leading zeros in the cipher text output. The output of the encryption algorithm is a variable of type long that is printed out as a hexadecimal string. Unfortunately, leading zeros are truncated, which caused this bug. To remedy this, the code was changed to ensure the leading zeros were printed out. This is accomplished by breaking the encryption output (which is of type long) into bytes. Each byte is checked and if it is less than 0x10, then a zero is printed, followed by the value of the byte. After this, the input plain text always matches the output plain text, as expected.

2. Unexpected GUI Actions

During encryption or decryption, the “Run” button is disabled so that the user cannot hit the button again while work is being done. Originally, the plan was to have what is called an activity indicator that would spin when work was being done during encryption or decryption. This plan had to be changed, however, due to how activity indicators are used. These indicators rely on the application using threads when running. Only when using threads can an activity monitor be enabled and spin. If the application does not use threads, the activity monitor will not run properly (it actually just does not work). Due to not finding an elegant way of having threads within this application, this design choice was changed. To show that the program was still doing work and to prevent the user from running the encryption or decryption method multiple times, the “Run” button is disabled at the start of these methods and enabled when these methods finish.

The virtual keyboard presented a few issues to the application as well. The biggest problem was determining how the application should handle the keyboard. By default, once the keyboard is on the screen, it does not go away. To get around this, the keyboard has to be programmatically told when to resign its “First Responder” or “current focus” status. This can be done in a number of ways, but only one way was chosen. When typing in the Message text-field, the user taps the return key to move to the Password text-field. There they tap the “Done” key to close the keyboard.

To handle the keyboard operations, a single function is used. When functions are called, they can be passed a sender ID (which allows the programmer to determine what GUI element called the function). After finding this out, the function can then match the action required depending on the field that called the function. This became a troublesome task, however, as there is no easy way to match the sender's ID to an ID of an element. Instead, Cocoa allows for each element to be assigned a tag number by the programmer. By looking at the sender's tag, the programmer can identify which sender called the function. After implementing this, the keyboard functionality can be easily handled in one function, rather than creating two separate functions.

V. CLOSING AND FUTURE CONSIDERATIONS

While people are becoming increasingly aware of the need for more stringent security measures throughout their daily lives, it can be very difficult to actually convince them to start using security applications or more secure practices. Most people forgo better security for ease of use. One such example of this is when people use the same password for multiple accounts. A recent article at *PC World* looked into this dangerous practice. In that article, the author, Carrie-Ann Skinner, gave several facts to back up her stance. According to a poll conducted by the hardware and software security vendor Sophos, “A third of web users have admitted to using the same password for a number of different websites.”

Therefore, the goal of this application is to make security accessible. Generally speaking, if people have to change their habits, it is much harder for them to start using an application. This is why the application does not have a way to send the message, but rather offers a cipher text that the user can copy into an SMS message or an email message, depending on what they prefer to use.

The objectives of this thesis were met and tested by a series of simulations using the iPhone simulator, an iPhone 3G, and an iPod Touch, all running the latest operating system (iPhone OS 3.0). The Blowfish application that was

created acts as a layer in the communication process by allowing a sender to encrypt a message with a passkey that they have shared with the receiver, then copy the cipher-text and send it to the receiver; the receiver can then take the cipher-text, paste it into the Blowfish application and decrypt it using the shared passkey. In completing this application, the objective of learning how to develop for the iPhone has also been completed.

While this method is easy to implement and it does not require the user to learn a new technology or enroll in another service, it is not as easy as it could be. There is a future enhancement that would make the application considerably more useable. This enhancement would incorporate a method that would allow the user to send their secure messages directly from the application, either by an SMS message, an email, or from a third party service that could be developed in conjunction with this application.

The first two methods could be improved by generating and sending the messages directly from the application instead of having to copy the cipher text and then enter the SMS application or the email application. While this is a great idea, it is impossible to implement in the current iPhone operating system. Currently, the iPhone operating system (as of OS 3.0) does not offer an SMS application programming interface (API). Since Apple does not offer an SMS API, there is no way to interact with the SMS application from another application. This prevents any application from creating a message and sending it through the SMS application.

The iPhone operating system does not have an email API, but it is possible to generate an email message and open it in the Mail application by using the "mailto" protocol. The "mailto" protocol is a Uniform Resource Locator (URL) that creates a message with certain attributes (who the message is to, the subject of the message and the message's contents, just to name a few) and opens that new message in the user's default mail client (the Mail application in the case of the iPhone). If used properly, this would present the user with a new email message where the subject is the cipher text that has been generated and the user can then input the recipient's email address and a subject if they wish.

The email method can also work for sending SMS messages as well. If a user has a cellular number, they have an email address that SMS messages can be emailed to in which they can receive the message on their device and the message will look and act like an SMS message. For example, if the recipient had cellular service with AT&T, a user could email the recipient an SMS message by sending an email to "xxxxxxxxxx@txt.att.net" where the first ten x's is the recipient's ten-digit cellular number. While this method could be helpful, it would require that the user know the recipient's cellular provider before they could email the recipient an SMS message. Even with this setback, the process could be streamlined, but would still require the user review the message and send it by using their Mail application. With that in mind, the application has been left open to where the user can choose to use either the email or SMS application on the iPhone.

Another possibility would be to setup a messaging service. This would require a server to direct all of the messages and setup by the user before they could start using it. They would have to set up an account with a username (or just make their username their phone number) so that the server would know which messages went to each account. With Apple's new push notification system implemented in OS 3.0, users could even be alerted when they received a new message. They could then run the application and download their message, which would in turn delete it off the server (thus keeping the safe practice of not storing the message on the server for a long time). This approach would require the user to change their habits and adopt a new method for sending messages to their contacts, which could prevent some users from adopting this new system.

Ideally, an API for SMS messaging on every phone operating system would exist that would allow third party developers to take advantage of this popular device-to-device messaging system. This would allow for better security in an insecure messaging market.

APPENDIX I

```
//
//  BlowfishCipher.h
//  Blowfish
//
//  Created by Todd Grooms on 6/27/09.
//

#import <Foundation/Foundation.h>

@interface BlowfishCipher : NSObject {
    unsigned long pBox_Gen[18];
    unsigned long sBox1_Gen[256];
    unsigned long sBox2_Gen[256];
    unsigned long sBox3_Gen[256];
    unsigned long sBox4_Gen[256];
}
- (void)initializeBlowfishWithPasskey:
    (char[])key andKeySize:(int)keyBytes;
- (void)blowfishEncrypt:(unsigned long *)dataLeft
    andData:(unsigned long *)dataRight;
- (void)blowfishDecrypt:(unsigned long *)dataLeft
    andData:(unsigned long *)dataRight;
- (unsigned long)F:(unsigned long)data;

@end
```

APPENDIX II

```
//
//  BlowfishCipher.m
//  Blowfish
//
//  Created by Todd Grooms on 6/27/09.
//

#import "BlowfishCipher.h"

// Initial P-Box with Hexadecimal Digits of Pi
static unsigned long pBox[] = {
    0x243f6a88L, 0x85a308d3L, 0x13198a2eL, 0x03707344L, 0xa4093822L,
    0x299f31d0L, 0x082efa98L, 0xec4e6c89L, 0x452821e6L, 0x38d01377L,
    0xbe5466cfL, 0x34e90c6cL, 0xc0ac29b7L, 0xc97c50ddL, 0x3f84d5b5L,
    0xb5470917L, 0x9216d5d9L, 0x8979fb1bL
};

// Initialize Four S-Boxes with Hexadecimal Digits of Pi
static unsigned long sBox1[] = {
    0xd1310ba6L, 0x98dfb5acL, 0x2ffd72dbL, 0xd01adfb7L, 0xb8e1afedL,
    0x6a267e96L, 0xba7c9045L, 0xf12c7f99L, 0x24a19947L, 0xb3916cf7L,
    0x0801f2e2L, 0x858efc16L, 0x636920d8L, 0x71574e69L, 0xa458fea3L,
    0xf4933d7eL, 0x0d95748fL, 0x728eb658L, 0x718bcd58L, 0x82154aeeL,
    0x7b54a41dL, 0xc25a59b5L, 0x9c30d539L, 0x2af26013L, 0xc5d1b023L,
    0x286085f0L, 0xca417918L, 0xb8db38efL, 0x8e79dcb0L, 0x603a180eL,
    0x6c9e0e8bL, 0xb01e8a3eL, 0xd71577c1L, 0xbd314b27L, 0x78af2fdaL,
    0x55605c60L, 0xe65525f3L, 0xaa55ab94L, 0x57489862L, 0x63e81440L,
    0x55ca396aL, 0x2aab10b6L, 0xb4cc5c34L, 0x1141e8ceL, 0xa15486afL,
    0x7c72e993L, 0xb3ee1411L, 0x636fbc2aL, 0x2ba9c55dL, 0x741831f6L,
    0xce5c3e16L, 0x9b87931eL, 0xafd6ba33L, 0x6c24cf5cL, 0x7a325381L,
    0x28958677L, 0x3b8f4898L, 0x6b4bb9afL, 0xc4bfe81bL, 0x66282193L,
    0x61d809ccL, 0xfb21a991L, 0x487cac60L, 0x5dec8032L, 0xef845d5dL,
    0xe98575b1L, 0xdc262302L, 0xeb651b88L, 0x23893e81L, 0xd396acc5L,
    0x0f6d6ff3L, 0x83f44239L, 0x2e0b4482L, 0xa4842004L, 0x69c8f04aL,
    0x9e1f9b5eL, 0x21c66842L, 0xf6e96c9aL, 0x670c9c61L, 0xabd388f0L,
    0x6a51a0d2L, 0xd8542f68L, 0x960fa728L, 0xab5133a3L, 0x6eef0b6cL,
    0x137a3be4L, 0xba3bf050L, 0x7efb2a98L, 0xa1f1651dL, 0x39af0176L,
    0x66ca593eL, 0x82430e88L, 0x8cee8619L, 0x456f9fb4L, 0x7d84a5c3L,
    0x3b8b5ebeL, 0xe06f75d8L, 0x85c12073L, 0x401a449fL, 0x56c16aa6L,
    0x4ed3aa62L, 0x363f7706L, 0x1bfedf72L, 0x429b023dL, 0x37d0d724L,
    0xd00a1248L, 0xdb0fead3L, 0x49f1c09bL, 0x075372c9L, 0x80991b7bL,
    0x25d479d8L, 0xf6e8def7L, 0xe3fe501aL, 0xb6794c3bL, 0x976ce0bdL,
    0x04c006baL, 0xc1a94fb6L, 0x409f60c4L, 0x5e5c9ec2L, 0x196a2463L,
    0x68fb6fafL, 0x3e6c53b5L, 0x1339b2ebL, 0x3b52ec6fL, 0x6dfc511fL,
    0x9b30952cL, 0xcc814544L, 0xaf5ebd09L, 0xbee3d004L, 0xde334afdL,
    0x660f2807L, 0x192e4bb3L, 0xc0cba857L, 0x45c8740fL, 0xd20b5f39L,
    0xb9d3fbdbL, 0x5579c0bdL, 0x1a60320aL, 0xd6a100c6L, 0x402c7279L,
    0x679f25feL, 0xfb1fa3ccL, 0x8ea5e9f8L, 0xdb3222f8L, 0x3c7516dfL,
    0xfd616b15L, 0x2f501ec8L, 0xad0552abL, 0x323db5faL, 0xfd238760L,
```

```
    0x53317b48L, 0x3e00df82L, 0x9e5c57bbL, 0xca6f8ca0L, 0x1a87562eL,
    0xdf1769dbL, 0xd542a8f6L, 0x287effc3L, 0xac6732c6L, 0x8c4f5573L,
    0x695b27b0L, 0xbbca58c8L, 0xe1ffa35dL, 0xb8f011a0L, 0x10fa3d98L,
    0xfd2183b8L, 0x4afcb56cL, 0x2dd1d35bL, 0x9a53e479L, 0xb6f84565L,
    0xd28e49bcL, 0x4bfb9790L, 0xe1ddf2daL, 0xa4cb7e33L, 0x62fb1341L,
    0xcee4c6e8L, 0xef20cadaL, 0x36774c01L, 0xd07e9efeL, 0x2bf11fb4L,
    0x95dbda4dL, 0xae909198L, 0xeaad8e71L, 0x6b93d5a0L, 0xd08ed1d0L,
    0xafc725e0L, 0x8e3c5b2fL, 0x8e7594b7L, 0x8ff6e2fbL, 0xf2122b64L,
    0x8888b812L, 0x900df01cL, 0x4fad5ea0L, 0x688fc31cL, 0xd1cff191L,
    0xb3a8c1adL, 0x2f2f2218L, 0xbe0e1777L, 0xea752dfeL, 0x8b021fa1L,
    0xe5a0cc0fL, 0xb56f74e8L, 0x18acf3d6L, 0xce89e299L, 0xb4a84fe0L,
    0xfd13e0b7L, 0x7cc43b81L, 0xd2ada8d9L, 0x165fa266L, 0x80957705L,
    0x93cc7314L, 0x211a1477L, 0xe6ad2065L, 0x77b5fa86L, 0xc75442f5L,
    0xfb9d35cfL, 0xebcdaf0cL, 0x7b3e89a0L, 0xd6411bd3L, 0xae1e7e49L,
    0x00250e2dL, 0x2071b35eL, 0x226800bbL, 0x57b8e0afL, 0x2464369bL,
    0xf009b91eL, 0x5563911dL, 0x59dfa6aaL, 0x78c14389L, 0xd95a537fL,
    0x207d5ba2L, 0x02e5b9c5L, 0x83260376L, 0x6295cfa9L, 0x11c81968L,
    0x4e734a41L, 0xb3472dcaL, 0x7b14a94aL, 0x1b510052L, 0x9a532915L,
    0xd60f573fL, 0xbc9bc6e4L, 0x2b60a476L, 0x81e67400L, 0x08ba6fb5L,
    0x571be91fL, 0xf296ec6bL, 0x2a0dd915L, 0xb6636521L, 0xe7b9f9b6L,
    0xff34052eL, 0xc5855664L, 0x53b02d5dL, 0xa99f8fa1L, 0x08ba4799L,
    0x6e85076aL
};

static unsigned long sBox2[] = {
    0x4b7a70e9L, 0xb5b32944L, 0xdb75092eL, 0xc4192623L, 0xad6ea6b0L,
    0x49a7df7dL, 0x9cee60b8L, 0x8fedb266L, 0xecaa8c71L, 0x699a17ffL,
    0x5664526cL, 0xc2b19ee1L, 0x193602a5L, 0x75094c29L, 0xa0591340L,
    0xe4183a3eL, 0x3f54989aL, 0x5b429d65L, 0x6b8fe4d6L, 0x99f73fd6L,
    0xa1d29c07L, 0xefe830f5L, 0x4d2d38e6L, 0xf0255dc1L, 0x4cdd2086L,
    0x8470eb26L, 0x6382e9c6L, 0x021ecc5eL, 0x09686b3fL, 0x3ebaefc9L,
    0x3c971814L, 0x6b6a70a1L, 0x687f3584L, 0x52a0e286L, 0xb79c5305L,
    0xaa500737L, 0x3e07841cL, 0x7fdeae5cL, 0x8e7d44ecL, 0x5716f2b8L,
    0xb03ada37L, 0xf0500c0dL, 0xf01c1f04L, 0x0200b3ffL, 0xae0cf51aL,
    0x3cb574b2L, 0x25837a58L, 0xdc0921bdL, 0xd19113f9L, 0x7ca92ff6L,
    0x94324773L, 0x22f54701L, 0x3ae5e581L, 0x37c2dadcL, 0xc8b57634L,
    0x9af3dda7L, 0xa9446146L, 0x0fd0030eL, 0xecc8c73eL, 0xa4751e41L,
    0xe238cd99L, 0x3bea0e2fL, 0x3280bba1L, 0x183eb331L, 0x4e548b38L,
    0x4f6db908L, 0x6f420d03L, 0xf60a04bfL, 0x2cb81290L, 0x24977c79L,
    0x5679b072L, 0xbcaf89afL, 0xde9a771fL, 0xd9930810L, 0xb38bae12L,
    0xdccf3f2eL, 0x5512721fL, 0x2e6b7124L, 0x501adde6L, 0x9f84cd87L,
    0x7a584718L, 0x7408da17L, 0xbc9f9abcL, 0xe94b7d8cL, 0xec7aec3aL,
    0xdb851dfaL, 0x63094366L, 0xc464c3d2L, 0xef1c1847L, 0x3215d908L,
    0xdd433b37L, 0x24c2ba16L, 0x12a14d43L, 0x2a65c451L, 0x50940002L,
    0x133ae4ddL, 0x71dff89eL, 0x10314e55L, 0x81ac77d6L, 0x5f11199bL,
    0x043556f1L, 0xd7a3c76bL, 0x3c11183bL, 0x5924a509L, 0xf28fe6edL,
    0x97f1fbfaL, 0x9ebabf2cL, 0x1e153c6eL, 0x86e34570L, 0xeae96fb1L,
    0x860e5e0aL, 0x5a3e2ab3L, 0x771fe71cL, 0x4e3d06faL, 0x2965dcb9L,
    0x99e71d0fL, 0x803e89d6L, 0x5266c825L, 0x2e4cc978L, 0x9c10b36aL,
    0xc6150ebaL, 0x94e2ea78L, 0xa5fc3c53L, 0x1e0a2df4L, 0xf2f74ea7L,
    0x361d2b3dL, 0x1939260fL, 0x19c27960L, 0x5223a708L, 0xf71312b6L,
    0xebadfe6eL, 0xeac31f66L, 0xe3bc4595L, 0xa67bc883L, 0xb17f37d1L,
    0x018cff28L, 0xc332ddefL, 0xbe6c5aa5L, 0x65582185L, 0x68ab9802L,
    0xeecea50fL, 0xdb2f953bL, 0x2aef7dadL, 0x5b6e2f84L, 0x1521b628L,
```

```
    0x29076170L, 0xecdd4775L, 0x619f1510L, 0x13cca830L, 0xeb61bd96L,
    0x0334fe1eL, 0xaa0363cfL, 0xb5735c90L, 0x4c70a239L, 0xd59e9e0bL,
    0xcbaade14L, 0xeecc86bcL, 0x60622ca7L, 0x9cab5cabL, 0xb2f3846eL,
    0x648b1eafL, 0x19bdf0caL, 0xa02369b9L, 0x655abb50L, 0x40685a32L,
    0x3c2ab4b3L, 0x319ee9d5L, 0xc021b8f7L, 0x9b540b19L, 0x875fa099L,
    0x95f7997eL, 0x623d7da8L, 0xf837889aL, 0x97e32d77L, 0x11ed935fL,
    0x16681281L, 0x0e358829L, 0xc7e61fd6L, 0x96dedfa1L, 0x7858ba99L,
    0x57f584a5L, 0x1b227263L, 0x9b83c3ffL, 0x1ac24696L, 0xcdb30aebL,
    0x532e3054L, 0x8fd948e4L, 0x6dbc3128L, 0x58ebf2efL, 0x34c6ffeaL,
    0xfe28ed61L, 0xee7c3c73L, 0x5d4a14d9L, 0xe864b7e3L, 0x42105d14L,
    0x203e13e0L, 0x45eee2b6L, 0xa3aaabeaL, 0xdb6c4f15L, 0xfacb4fd0L,
    0xc742f442L, 0xef6abbb5L, 0x654f3b1dL, 0x41cd2105L, 0xd81e799eL,
    0x86854dc7L, 0xe44b476aL, 0x3d816250L, 0xcf62a1f2L, 0x5b8d2646L,
    0xfc8883a0L, 0xc1c7b6a3L, 0x7f1524c3L, 0x69cb7492L, 0x47848a0bL,
    0x5692b285L, 0x095bbf00L, 0xad19489dL, 0x1462b174L, 0x23820e00L,
    0x58428d2aL, 0x0c55f5eaL, 0x1dadf43eL, 0x233f7061L, 0x3372f092L,
    0x8d937e41L, 0xd65fecf1L, 0x6c223bdbL, 0x7cde3759L, 0xcbee7460L,
    0x4085f2a7L, 0xce77326eL, 0xa6078084L, 0x19f8509eL, 0xe8efd855L,
    0x61d99735L, 0xa969a7aaL, 0xc50c06c2L, 0x5a04abfcL, 0x800bcadcL,
    0x9e447a2eL, 0xc3453484L, 0xfdd56705L, 0x0e1e9ec9L, 0xdb73dbd3L,
    0x105588cdL, 0x675fda79L, 0xe3674340L, 0xc5c43465L, 0x713e38d8L,
    0x3d28f89eL, 0xf16dff20L, 0x153e21e7L, 0x8fb03d4aL, 0xe6e39f2bL,
    0xdb83adf7L
};

static unsigned long sBox3[] = {
    0xe93d5a68L, 0x948140f7L, 0xf64c261cL, 0x94692934L, 0x411520f7L,
    0x7602d4f7L, 0xbcf46b2eL, 0xd4a20068L, 0xd4082471L, 0x3320f46aL,
    0x43b7d4b7L, 0x500061afL, 0x1e39f62eL, 0x97244546L, 0x14214f74L,
    0xbf8b8840L, 0x4d95fc1dL, 0x96b591afL, 0x70f4ddd3L, 0x66a02f45L,
    0xbfbc09ecL, 0x03bd9785L, 0x7fac6dd0L, 0x31cb8504L, 0x96eb27b3L,
    0x55fd3941L, 0xda2547e6L, 0xabca0a9aL, 0x28507825L, 0x530429f4L,
    0x0a2c86daL, 0xe9b66dfbL, 0x68dc1462L, 0xd7486900L, 0x680ec0a4L,
    0x27a18deeL, 0x4f3ffea2L, 0xe887ad8cL, 0xb58ce006L, 0x7af4d6b6L,
    0xaace1e7cL, 0xd3375fecL, 0xce78a399L, 0x406b2a42L, 0x20fe9e35L,
    0xd9f385b9L, 0xee39d7abL, 0x3b124e8bL, 0x1dc9faf7L, 0x4b6d1856L,
    0x26a36631L, 0xeae397b2L, 0x3a6efa74L, 0xdd5b4332L, 0x6841e7f7L,
    0xca7820fbL, 0xfb0af54eL, 0xd8feb397L, 0x454056acL, 0xba489527L,
    0x55533a3aL, 0x20838d87L, 0xfe6ba9b7L, 0xd096954bL, 0x55a867bcL,
    0xa1159a58L, 0xcca92963L, 0x99e1db33L, 0xa62a4a56L, 0x3f3125f9L,
    0x5ef47e1cL, 0x9029317cL, 0xfdf8e802L, 0x04272f70L, 0x80bb155cL,
    0x05282ce3L, 0x95c11548L, 0xe4c66d22L, 0x48c1133fL, 0xc70f86dcL,
    0x07f9c9eeL, 0x41041f0fL, 0x404779a4L, 0x5d886e17L, 0x325f51ebL,
    0xd59bc0d1L, 0xf2bcc18fL, 0x41113564L, 0x257b7834L, 0x602a9c60L,
    0xdff8e8a3L, 0x1f636c1bL, 0x0e12b4c2L, 0x02e1329eL, 0xaf664fd1L,
    0xcad18115L, 0x6b2395e0L, 0x333e92e1L, 0x3b240b62L, 0xeebeb922L,
    0x85b2a20eL, 0xe6ba0d99L, 0xde720c8cL, 0x2da2f728L, 0xd0127845L,
    0x95b794fdL, 0x647d0862L, 0xe7ccf5f0L, 0x5449a36fL, 0x877d48faL,
    0xc39dfd27L, 0xf33e8d1eL, 0x0a476341L, 0x992eff74L, 0x3a6f6eabL,
    0xf4f8fd37L, 0xa812dc60L, 0xa1ebddf8L, 0x991be14cL, 0xdb6e6b0dL,
    0xc67b5510L, 0x6d672c37L, 0x2765d43bL, 0xdcd0e804L, 0xf1290dc7L,
    0xcc00ffa3L, 0xb5390f92L, 0x690fed0bL, 0x667b9ffbL, 0xcedb7d9cL,
    0xa091cf0bL, 0xd9155ea3L, 0xbb132f88L, 0x515bad24L, 0x7b9479bfL,
    0x763bd6ebL, 0x37392eb3L, 0xcc115979L, 0x8026e297L, 0xf42e312dL,
```

```
    0x6842ada7L, 0xc66a2b3bL, 0x12754cccL, 0x782ef11cL, 0x6a124237L,
    0xb79251e7L, 0x06a1bbe6L, 0x4bfb6350L, 0x1a6b1018L, 0x11caedfaL,
    0x3d25bdd8L, 0xe2e1c3c9L, 0x44421659L, 0x0a121386L, 0xd90cec6eL,
    0xd5abea2aL, 0x64af674eL, 0xda86a85fL, 0xbebfe988L, 0x64e4c3feL,
    0x9dbc8057L, 0xf0f7c086L, 0x60787bf8L, 0x6003604dL, 0xd1fd8346L,
    0xf6381fb0L, 0x7745ae04L, 0xd736fcccL, 0x83426b33L, 0xf01eab71L,
    0xb0804187L, 0x3c005e5fL, 0x77a057beL, 0xbde8ae24L, 0x55464299L,
    0xbf582e61L, 0x4e58f48fL, 0xf2ddfda2L, 0xf474ef38L, 0x8789bdc2L,
    0x5366f9c3L, 0xc8b38e74L, 0xb475f255L, 0x46fcd9b9L, 0x7aeb2661L,
    0x8b1ddf84L, 0x846a0e79L, 0x915f95e2L, 0x466e598eL, 0x20b45770L,
    0x8cd55591L, 0xc902de4cL, 0xb90bace1L, 0xbb8205d0L, 0x11a86248L,
    0x7574a99eL, 0xb77f19b6L, 0xe0a9dc09L, 0x662d09a1L, 0xc4324633L,
    0xe85a1f02L, 0x09f0be8cL, 0x4a99a025L, 0x1d6efe10L, 0x1ab93d1dL,
    0x0ba5a4dfL, 0xa186f20fL, 0x2868f169L, 0xdcb7da83L, 0x573906feL,
    0xa1e2ce9bL, 0x4fcd7f52L, 0x50115e01L, 0xa70683faL, 0xa002b5c4L,
    0x0de6d027L, 0x9af88c27L, 0x773f8641L, 0xc3604c06L, 0x61a806b5L,
    0xf0177a28L, 0xc0f586e0L, 0x006058aaL, 0x30dc7d62L, 0x11e69ed7L,
    0x2338ea63L, 0x53c2dd94L, 0xc2c21634L, 0xbbcbee56L, 0x90bcb6deL,
    0xebfc7da1L, 0xce591d76L, 0x6f05e409L, 0x4b7c0188L, 0x39720a3dL,
    0x7c927c24L, 0x86e3725fL, 0x724d9db9L, 0x1ac15bb4L, 0xd39eb8fcL,
    0xed545578L, 0x08fca5b5L, 0xd83d7cd3L, 0x4dad0fc4L, 0x1e50ef5eL,
    0xb161e6f8L, 0xa28514d9L, 0x6c51133cL, 0x6fd5c7e7L, 0x56e14ec4L,
    0x362abfceL, 0xddc6c837L, 0xd79a3234L, 0x92638212L, 0x670efa8eL,
    0x406000e0L
};

static unsigned long sBox4[] = {
    0x3a39ce37L, 0xd3faf5cfL, 0xabc27737L, 0x5ac52d1bL, 0x5cb0679eL,
    0x4fa33742L, 0xd3822740L, 0x99bc9bbeL, 0xd5118e9dL, 0xbf0f7315L,
    0xd62d1c7eL, 0xc700c47bL, 0xb78c1b6bL, 0x21a19045L, 0xb26eb1beL,
    0x6a366eb4L, 0x5748ab2fL, 0xbc946e79L, 0xc6a376d2L, 0x6549c2c8L,
    0x530ff8eeL, 0x468dde7dL, 0xd5730a1dL, 0x4cd04dc6L, 0x2939bbdbL,
    0xa9ba4650L, 0xac9526e8L, 0xbe5ee304L, 0xa1fad5f0L, 0x6a2d519aL,
    0x63ef8ce2L, 0x9a86ee22L, 0xc089c2b8L, 0x43242ef6L, 0xa51e03aaL,
    0x9cf2d0a4L, 0x83c061baL, 0x9be96a4dL, 0x8fe51550L, 0xba645bd6L,
    0x2826a2f9L, 0xa73a3ae1L, 0x4ba99586L, 0xef5562e9L, 0xc72fefd3L,
    0xf752f7daL, 0x3f046f69L, 0x77fa0a59L, 0x80e4a915L, 0x87b08601L,
    0x9b09e6adL, 0x3b3ee593L, 0xe990fd5aL, 0x9e34d797L, 0x2cf0b7d9L,
    0x022b8b51L, 0x96d5ac3aL, 0x017da67dL, 0xd1cf3ed6L, 0x7c7d2d28L,
    0x1f9f25cfL, 0xadf2b89bL, 0x5ad6b472L, 0x5a88f54cL, 0xe029ac71L,
    0xe019a5e6L, 0x47b0acfdL, 0xed93fa9bL, 0xe8d3c48dL, 0x283b57ccL,
    0xf8d56629L, 0x79132e28L, 0x785f0191L, 0xed756055L, 0xf7960e44L,
    0xe3d35e8cL, 0x15056dd4L, 0x88f46dbaL, 0x03a16125L, 0x0564f0bdL,
    0xc3eb9e15L, 0x3c9057a2L, 0x97271aecL, 0xa93a072aL, 0x1b3f6d9bL,
    0x1e6321f5L, 0xf59c66fbL, 0x26dcf319L, 0x7533d928L, 0xb155fdf5L,
    0x03563482L, 0x8aba3cbbL, 0x28517711L, 0xc20ad9f8L, 0xabcc5167L,
    0xccad925fL, 0x4de81751L, 0x3830dc8eL, 0x379d5862L, 0x9320f991L,
    0xea7a90c2L, 0xfb3e7bceL, 0x5121ce64L, 0x774fbe32L, 0xa8b6e37eL,
    0xc3293d46L, 0x48de5369L, 0x6413e680L, 0xa2ae0810L, 0xdd6db224L,
    0x69852dfdL, 0x09072166L, 0xb39a460aL, 0x6445c0ddL, 0x586cdecfL,
    0x1c20c8aeL, 0x5bbef7ddL, 0x1b588d40L, 0xccd2017fL, 0x6bb4e3bbL,
    0xdda26a7eL, 0x3a59ff45L, 0x3e350a44L, 0xbcb4cdd5L, 0x72eacea8L,
    0xfa6484bbL, 0x8d6612aeL, 0xbf3c6f47L, 0xd29be463L, 0x542f5d9eL,
    0xaec2771bL, 0xf64e6370L, 0x740e0d8dL, 0xe75b1357L, 0xf8721671L,
```

```
    0xaf537d5dL, 0x4040cb08L, 0x4eb4e2ccL, 0x34d2466aL, 0x0115af84L,
    0xe1b00428L, 0x95983a1dL, 0x06b89fb4L, 0xce6ea048L, 0x6f3f3b82L,
    0x3520ab82L, 0x011a1d4bL, 0x277227f8L, 0x611560b1L, 0xe7933fdcL,
    0xbb3a792bL, 0x344525bdL, 0xa08839e1L, 0x51ce794bL, 0x2f32c9b7L,
    0xa01fbac9L, 0xe01cc87eL, 0xbcc7d1f6L, 0xcf0111c3L, 0xa1e8aac7L,
    0x1a908749L, 0xd44fbd9aL, 0xd0dadecbL, 0xd50ada38L, 0x0339c32aL,
    0xc6913667L, 0x8df9317cL, 0xe0b12b4fL, 0xf79e59b7L, 0x43f5bb3aL,
    0xf2d519ffL, 0x27d9459cL, 0xbf97222cL, 0x15e6fc2aL, 0x0f91fc71L,
    0x9b941525L, 0xfae59361L, 0xceb69cebL, 0xc2a86459L, 0x12baa8d1L,
    0xb6c1075eL, 0xe3056a0cL, 0x10d25065L, 0xcb03a442L, 0xe0ec6e0eL,
    0x1698db3bL, 0x4c98a0beL, 0x3278e964L, 0x9f1f9532L, 0xe0d392dfL,
    0xd3a0342bL, 0x8971f21eL, 0x1b0a7441L, 0x4ba3348cL, 0xc5be7120L,
    0xc37632d8L, 0xdf359f8dL, 0x9b992f2eL, 0xe60b6f47L, 0x0fe3f11dL,
    0xe54cda54L, 0x1edad891L, 0xce6279cfL, 0xcd3e7e6fL, 0x1618b166L,
    0xfd2c1d05L, 0x848fd2c5L, 0xf6fb2299L, 0xf523f357L, 0xa6327623L,
    0x93a83531L, 0x56cccd02L, 0xacf08162L, 0x5a75ebb5L, 0x6e163697L,
    0x88d273ccL, 0xde966292L, 0x81b949d0L, 0x4c50901bL, 0x71c65614L,
    0xe6c6c7bdL, 0x327a140aL, 0x45e1d006L, 0xc3f27b9aL, 0xc9aa53fdL,
    0x62a80f00L, 0xbb25bfe2L, 0x35bdd2f6L, 0x71126905L, 0xb2040222L,
    0xb6cbcf7cL, 0xcd769c2bL, 0x53113ec0L, 0x1640e3d3L, 0x38abbd60L,
    0x2547adf0L, 0xba38209cL, 0xf746ce76L, 0x77afa1c5L, 0x20756060L,
    0x85cbfe4eL, 0x8ae88dd8L, 0x7aaaf9b0L, 0x4cf9aa7eL, 0x1948c25cL,
    0x02fb8a8cL, 0x01c36ae4L, 0xd6ebe1f9L, 0x90d4f869L, 0xa65cdea0L,
    0x3f09252dL, 0xc208e69fL, 0xb74e6132L, 0xce77e25bL, 0x578fdfe3L,
    0x3ac372e6L
};

@implementation BlowfishCipher

- (void)initializeBlowfishWithPasskey:(char[])key andKeySize:(int)keyBytes {
    // Step 1 of subkey creation is initializing pBox and sBoxes with hex Pi
    //digits
    //  -> This is taken care of in the header file.
    /* Step 2 of subkey creation:
     * XOR P1 with the first 32 bits of the key, XOR P2 with the second 32-bits
     * of the key, and so on for all bits of the key (possibly up to P14).
     * Repeatedly cycle through the key bits until the entire P-array has been
     * XORed with key bits. (For every short key, there is at least one
     * equivalent longer key; for example, if A is a 64-bit key,
     * then AA, AAA, etc., are equivalent keys.)
     */
    unsigned long data;
    int keyByteCounter = 0;
    int iterator;
    for(iterator = 0; iterator < 18; iterator++) {
      data = 0x00000000;
      for(int k = 0; k < 4; k++) {
          data = (data << 8) | key[keyByteCounter];
          keyByteCounter++;
          if(keyByteCounter >= keyBytes)
              keyByteCounter = 0;
      }
      pBox_Gen[iterator] = pBox[iterator] ^ data;
    }
```

```
for(iterator = 0; iterator < 256; iterator++) {
  sBox1_Gen[iterator] = sBox1[iterator];
}

for(iterator = 0; iterator < 256; iterator++) {
  sBox2_Gen[iterator] = sBox2[iterator];
}

for(iterator = 0; iterator < 256; iterator++) {
  sBox3_Gen[iterator] = sBox3[iterator];
}

for(iterator = 0; iterator < 256; iterator++) {
  sBox4_Gen[iterator] = sBox4[iterator];
}
/* Step 3 of subkey creation:
 * Encrypt the all-zero string with the Blowfish algorithm, using the
 * subkeys described in steps (1) and (2).
 *
 * Step 4 of subkey creation:
 * Replace P1 and P2 with the output of step (3).
 *
 * Step 5 of subkey creation:
 * Encrypt the output of step (3) using the Blowfish algorithm with the
 * modified subkeys.
 *
 * Step 6 of subkey creation:
 * Replace P3 and P4 with the output of step (5).
 *
 * Step 7 of subkey creation:
 * Continue the process, replacing all entries of the P- array, and then
 * all four S-boxes in order, with the output of the continuously-changing
 * Blowfish algorithm.
 */
unsigned long dataLeft = 0x00000000, dataRight = 0x00000000;
// pBox
for(int iterator = 0; iterator < 18; iterator += 2) {
  [self blowfishEncrypt:&dataLeft andData:&dataRight];
  pBox_Gen[iterator] = dataLeft;
  pBox_Gen[iterator + 1] = dataRight;
}

// sBox1
for(iterator = 0; iterator < 256; iterator += 2) {
  [self blowfishEncrypt:&dataLeft andData:&dataRight];
  sBox1_Gen[iterator] = dataLeft;
  sBox1_Gen[iterator + 1] = dataRight;
}
// sBox2
for(iterator = 0; iterator < 256; iterator += 2) {
  [self blowfishEncrypt:&dataLeft andData:&dataRight];
  sBox2_Gen[iterator] = dataLeft;
  sBox2_Gen[iterator + 1] = dataRight;
}
```

```
    // sBox3
    for(iterator = 0; iterator < 256; iterator += 2) {
      [self blowfishEncrypt:&dataLeft andData:&dataRight];
      sBox3_Gen[iterator] = dataLeft;
      sBox3_Gen[iterator + 1] = dataRight;
    }
    // sBox4
    for(iterator = 0; iterator < 256; iterator += 2) {
      [self blowfishEncrypt:&dataLeft andData:&dataRight];
      sBox4_Gen[iterator] = dataLeft;
      sBox4_Gen[iterator + 1] = dataRight;
    }
}

/* Blowfish F Function */
- (unsigned long)F:(unsigned long)data {
    // F-Function.
    // data is a 32 bit value; [ 31 | 30 | 29 | ... | 2 | 1 | 0 ]
    unsigned short a, b, c, d;
    unsigned long output;

    d = data & 0x00FF; //d = bits 7 to 0
    data >>= 8;

    c = data & 0x00FF; //c = bits 15 to 8
    data >>= 8;

    b = data & 0x00FF; //b = bits 23 to 16
    data >>= 8;

    a = data & 0x00FF; //a = bits 31 to 24

    //output = ((S1,a + S2,b mod 2^32) XOR S3,c) + S4,d mod 2^32
    output = ((sBox1_Gen[a] + sBox2_Gen[b]) ^ sBox3_Gen[c]) + sBox4_Gen[d];

    return output;
}

/* Blowfish Encryption Function */
- (void)blowfishEncrypt:(unsigned long *)dataLeft
  andData:(unsigned long *)dataRight {
    unsigned long dataTemp;

    for(int iterator = 0; iterator < 16; iterator++) {
      *dataLeft = *dataLeft ^ pBox_Gen[iterator];
      *dataRight = [self F:*dataLeft] ^ *dataRight;

      // Switch left leg with right leg
      dataTemp = *dataLeft;
      *dataLeft = *dataRight;
      *dataRight = dataTemp;
    }
```

```
    // Switch legs around again (above for loop switches legs an extra time;
    //   this step reverses them back)
    dataTemp = *dataLeft;
    *dataLeft = *dataRight;
    *dataRight = dataTemp;

    *dataRight = *dataRight ^ pBox_Gen[16];
    *dataLeft = *dataLeft ^ pBox_Gen[17];
}

/* Blowfish Decryption Function */
- (void)blowfishDecrypt:(unsigned long *)dataLeft
  andData:(unsigned long *)dataRight {
    unsigned long dataTemp;

    for(int iterator = 17; iterator > 1; iterator--) {
      *dataLeft = *dataLeft ^ pBox_Gen[iterator];
      *dataRight = [self F:*dataLeft] ^ *dataRight;

      // Switch left leg with right leg
      dataTemp = *dataLeft;
      *dataLeft = *dataRight;
      *dataRight = dataTemp;
    }

    // Switch legs around again (above for loop switches legs an extra time;
    //   this step reverses them back)
    dataTemp = *dataLeft;
    *dataLeft = *dataRight;
    *dataRight = dataTemp;

    *dataRight = *dataRight ^ pBox_Gen[1];
    *dataLeft = *dataLeft ^ pBox_Gen[0];
}
@end
```

APPENDIX III

```
//
//  BlowfishViewController.h
//  Blowfish
//
//  Created by Todd Grooms on 6/27/09.
//

#define kEncryptSegment 0
#define kDecryptSegment 1

#import <UIKit/UIKit.h>

@interface BlowfishViewController : UIViewController {
     IBOutlet UITextField *messageTextField;
     IBOutlet UITextField *passwordTextField;
     IBOutlet UISegmentedControl *methodControl;
     IBOutlet UITextView *resultsTextView;
     IBOutlet UIActivityIndicatorView *activityIndicator;
     IBOutlet UIButton *runButton;
}
@property(nonatomic, retain) UITextField *messageTextField;
@property(nonatomic, retain) UITextField *passwordTextField;
@property(nonatomic, retain) UISegmentedControl *methodControl;
@property(nonatomic, retain) UITextView *resultsTextView;
@property(nonatomic, retain) UIActivityIndicatorView *activityIndicator;
@property(nonatomic, retain) UIButton *runButton;
- (IBAction)editingFieldDone:(id)sender;
- (IBAction)runButtonPressed:(id)sender;
- (unsigned long)hexCharToLong:(char)ch;
- (void)performOperation;

@end
```

APPENDIX IV

```
//
//  BlowfishViewController.m
//  Blowfish
//
//  Created by Todd Grooms on 6/27/09.
//

#import "BlowfishViewController.h"
#import "BlowfishCipher.h"

@implementation BlowfishViewController
@synthesize messageTextField;
@synthesize passwordTextField;
@synthesize methodControl;
@synthesize resultsTextView;
@synthesize activityIndicator;
@synthesize runButton;

/* Convert hexadecimal character to long value */
- (unsigned long)hexCharToLong:(char)ch {
    unsigned long longValue;
    switch(ch) {
        case '0':
            longValue = 0x00000000;
            break;
        case '1':
            longValue = 0x00000001;
            break;
        case '2':
            longValue = 0x00000002;
            break;
        case '3':
            longValue = 0x00000003;
            break;
        case '4':
            longValue = 0x00000004;
            break;
        case '5':
            longValue = 0x00000005;
            break;
        case '6':
            longValue = 0x00000006;
            break;
        case '7':
            longValue = 0x00000007;
            break;
        case '8':
            longValue = 0x00000008;
            break;
```

```
        case '9':
            longValue = 0x00000009;
            break;
        case 'a':
            longValue = 0x0000000A;
            break;
        case 'b':
            longValue = 0x0000000B;
            break;
        case 'c':
            longValue = 0x0000000C;
            break;
        case 'd':
            longValue = 0x0000000D;
            break;
        case 'e':
            longValue = 0x0000000E;
            break;
        case 'f':
            longValue = 0x0000000F;
        default:
            break;
    }

    return longValue;
}

/* User taps 'Return' or 'Done' in virtual keyboard */
- (void)editingFieldDone:(id)sender {
    switch([sender tag]) {
        case 1:   //Finished editing messageTextField
            [passwordTextField  becomeFirstResponder];
            break;
        case 2:   //Finished editing passwordTextField
            [passwordTextField resignFirstResponder];
            break;
    }
}
```

```
/* User is running Blowfish */
- (void)runButtonPressed:(id)sender {
    if(messageTextField.text.length == 0) {
        // Error: Message field is blank
        NSInteger segment = methodControl.selectedSegmentIndex;
        NSString *method = [[NSString alloc] initWithString:@"The message field
is blank.  There is nothing to encrypt."];

        if(segment == kDecryptSegment) {
            method = @"The message field is blank.  There is nothing to
decrypt.";
        }
        UIAlertView *alert = [[UIAlertView alloc]
                              initWithTitle:@"Message Length Error"
                              message:method
                              delegate:nil
                              cancelButtonTitle:@"Close"
                              otherButtonTitles:nil];
        [alert show];
        [alert release];
        [method release];
        resultsTextView.text = @"";
    }
    else if(passwordTextField.text.length > 56) {
        // Error: Password is too long
        UIAlertView *alert = [[UIAlertView alloc]
                              initWithTitle:@"Password Length Error"
                              message:@"The password you have entered is too long.
The password must be 56 characters or less."
                              delegate:nil
                              cancelButtonTitle:@"Close"
                              otherButtonTitles:nil];
        [alert show];
        [alert release];
        resultsTextView.text = @"";
    }
    else {
        // Good to Run
        [runButton setEnabled:NO];
        [activityIndicator startAnimating];

        [self performOperation];
    }
}

/* Runs User's Selection: Either Encrypt or Decrypt Algorithm */
- (void)performOperation {
    // Prepare message
    int messageBytes = messageTextField.text.length;
    char message[messageBytes];
    strncpy(message, [messageTextField.text UTF8String], messageBytes);
```

```
    // Prepare password
    int passkeyLength = passwordTextField.text.length, passkeyBytes;
    if(passkeyLength == 0)
        passkeyBytes = 1;
    else
        passkeyBytes = passkeyLength;
    char passkey[passkeyBytes];
    if(passkeyLength > 0)
        strncpy(passkey, [passwordTextField.text UTF8String], passkeyBytes);
    else
        passkey[0] = 0x00;

    // Initialize an Instance of the Blowfish Cipher
    BlowfishCipher *bc = [[BlowfishCipher alloc] init];
    [bc initializeBlowfishWithPasskey:passkey andKeySize:passkeyBytes];

    // Result that needs to be displayed to user
    NSMutableString *result = [[NSMutableString alloc] initWithString:@""];

    //Find which method is selected
    NSInteger segment = methodControl.selectedSegmentIndex;
    if(segment == kEncryptSegment) {
        // Encrypt
        if(passwordTextField.text.length < 4) {
            // Warning: Password length is less than four
            UIAlertView *alert = [[UIAlertView alloc]
                                  initWithTitle:@"Password Length Warning"
                                  message:@"The password length is less than four
characters.  The message will still be encrypted, but it is a very weak
encryption.  Please use a password that contains at least four characters in
the future."
                                  delegate:nil
                                  cancelButtonTitle:@"Close"
                                  otherButtonTitles:nil];
            [alert show];
            [alert release];
        }
        for(int iterator = 0; iterator < messageBytes; iterator += 8) {
            // Obtain xL and xR values from input message
            unsigned long xL = 0x00000000, xR = 0x00000000;
            for(int counter = 0; counter < 3; counter++) {
                if(iterator + counter < messageBytes)
                    xL = xL | message[iterator + counter];
                xL <<= 8;
            }
            if(iterator + 3 < messageBytes)
                xL = xL | message[iterator + 3];

            for(int counter = 4; counter < 7; counter++) {
                if(iterator + counter < messageBytes)
                    xR = xR | message[iterator + counter];
                xR <<= 8;
            }
            if(iterator + 7 < messageBytes)
```

```
            xR = xR | message[iterator + 7];

        // Encrypt this batch of data
        [bc blowfishEncrypt:&xL andData:&xR];

        // Mask off appropriate bits and convert long values to strings
        unsigned long mask1 = 0xFF000000, mask2 = 0x00FF0000;
        unsigned long mask3 = 0x0000FF00, mask4 = 0x000000FF;
        unsigned long temp;

        // Also avoid losing leading 0's by manually printing them out when
        //   long values < 0x10
        temp = xL & mask1;
        temp >>= 24;
        if(temp < 0x10)
            [result appendString:@"0"];
        [result appendString:[NSString stringWithFormat:@"%x", temp]];
        temp = xL & mask2;
        temp >>= 16;
        if(temp < 0x10)
            [result appendString:@"0"];
        [result appendString:[NSString stringWithFormat:@"%x", temp]];
        temp = xL & mask3;
        temp >>= 8;
        if(temp < 0x10)
            [result appendString:@"0"];
        [result appendString:[NSString stringWithFormat:@"%x", temp]];
        temp = xL & mask4;
        if(temp < 0x10)
            [result appendString:@"0"];
        [result appendString:[NSString stringWithFormat:@"%x", temp]];

        temp = xR & mask1;
        temp >>= 24;
        if(temp < 0x10)
            [result appendString:@"0"];
        [result appendString:[NSString stringWithFormat:@"%x", temp]];
        temp = xR & mask2;
        temp >>= 16;
        if(temp < 0x10)
            [result appendString:@"0"];
        [result appendString:[NSString stringWithFormat:@"%x", temp]];
        temp = xR & mask3;
        temp >>= 8;
        if(temp < 0x10)
            [result appendString:@"0"];
        [result appendString:[NSString stringWithFormat:@"%x", temp]];
        temp = xR & mask4;
        if(temp < 0x10)
            [result appendString:@"0"];
        [result appendString:[NSString stringWithFormat:@"%x", temp]];
    }
    // Output resulting cipher-text
    resultsTextView.text = result;
```

```
}
else if(segment == kDecryptSegment) {
    // Decrypt
    for(int iterator = 0; iterator < messageBytes; iterator += 16) {
        // Obtain xL and xR values from input message
        unsigned long xL = 0x00000000, xR = 0x00000000;
        for(int counter = 0; counter < 7; counter++) {
            xL = xL | [self hexCharToLong:message[iterator + counter]];
            xL <<= 4;
        }
        xL = xL | [self hexCharToLong:message[iterator + 7]];

        for(int counter = 8; counter < 15; counter++) {
            xR = xR | [self hexCharToLong:message[iterator + counter]];
            xR <<= 4;
        }
        xR = xR | [self hexCharToLong:message[iterator + 15]];

        // Decrypt this batch of data
        [bc blowfishDecrypt:&xL andData:&xR];

        // Mask off appropriate bits and convert long values to characters
        //   (Hex long values are actually Hex ASCII)
        unsigned long mask1 = 0xFF000000, mask2 = 0x00FF0000;
        unsigned long mask3 = 0x0000FF00, mask4 = 0x000000FF;
        unsigned long temp;

        temp = xL & mask1;
        temp >>= 24;
        [result appendString:[NSString stringWithFormat:@"%c", temp]];
        temp = xL & mask2;
        temp >>= 16;
        [result appendString:[NSString stringWithFormat:@"%c", temp]];
        temp = xL & mask3;
        temp >>= 8;
        [result appendString:[NSString stringWithFormat:@"%c", temp]];
        temp = xL & mask4;
        [result appendString:[NSString stringWithFormat:@"%c", temp]];

        temp = xR & mask1;
        temp >>= 24;
        [result appendString:[NSString stringWithFormat:@"%c", temp]];
        temp = xR & mask2;
        temp >>= 16;
        [result appendString:[NSString stringWithFormat:@"%c", temp]];
        temp = xR & mask3;
        temp >>= 8;
        [result appendString:[NSString stringWithFormat:@"%c", temp]];
        temp = xR & mask4;
        [result appendString:[NSString stringWithFormat:@"%c", temp]];
    }
    // Output resulting plain-text
    resultsTextView.text = result;
}
```

```
    else {
        UIAlertView *alert = [[UIAlertView alloc]
                              initWithTitle:@"Method Error"
                              message:@"An error as occurred; neither method has
been chosen."
                              delegate:nil
                              cancelButtonTitle:@"Close"
                              otherButtonTitles:nil];
        [alert show];
        [alert release];
    }
    [bc release];
    [result release];
    [activityIndicator stopAnimating];
    [runButton setEnabled:YES];
}

// Implement viewDidLoad to do additional setup after loading the view,
//   typically from a nib.
- (void)viewDidLoad {
    UIImage *buttonImageNormal = [UIImage imageNamed:@"whiteButton.png"];
    UIImage *stretchableButtonImageNormal = [buttonImageNormal
        stretchableImageWithLeftCapWidth:12 topCapHeight:0];
    [runButton setBackgroundImage:stretchableButtonImageNormal
        forState:UIControlStateNormal];

    UIImage *buttonImagePressed = [UIImage imageNamed:@"blueButton.png"];
    UIImage *stretchableButtonImagePressed = [buttonImagePressed
        stretchableImageWithLeftCapWidth:12 topCapHeight:0];
    [runButton setBackgroundImage:stretchableButtonImagePressed
        forState:UIControlStateHighlighted];

     [super viewDidLoad];
}

- (void)didReceiveMemoryWarning {
     [super didReceiveMemoryWarning];
}

- (void)dealloc {
    [messageTextField release];
    [passwordTextField release];
    [methodControl release];
    [resultsTextView release];
    [activityIndicator release];
    [runButton release];
    [super dealloc];
}

@end
```

LIST OF REFERENCES

"Gartner Says Worldwide Smartphone Sales Reached Its Lowest Growth Rate With 3.7 Per Cent Increase in Fourth Quarter of 2008," *Gartner*, 11 March 2009, available from
http://www.gartner.com/it/page.jsp?id=910112
accessed 20 June 2009.

Enck, William, Patrick Traynor, Patrick McDaniel, Thomas La Porta. 2005. Exploiting Open Functionality in SMS-Capable Cellular Networks.
ACM Conference on Computer and Communications Security, available from
http://www.smsanalysis.org/

John Leyden, "SMS security risks highlighted by Friends Reunited hacking case: But you knew that, didn't you?," *The Register*, 27 November 2002, available from
http://www.theregister.co.uk/2002/11/27/sms_security_risks_highlighted_by/
accessed 20 June 2009.

Lohr, Steve. 2009. "Smartphone Rises Fast From Gadget to Necessity." *The New York Times* (New York City, New York). June 9.

Mark, Dave. Jeff LaMarche. 2009. *Beginning iPhone Development: Exploring the iPhone SDK*. Springer-Verlag New York, Inc.

Sandstrom, Gustav. 2009. "Handset Sales Drop, Though Smart Phones Sell Well." *The Wall Street Journal* (New York City, New York). May 21.

Bruce Schneier, "Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)," *The Blowfish Encryption Algorithm*, December 1993, available from
http://www.schneier.com/paper-blowfish-fse.html
accessed 20 June 2009.

Carrie-Ann Skinner, "One-Third Use a Single Password for Everything," *PC World*, 11 March 2009, available from
http://www.pcworld.com/businesscenter/article/161078/
onethird_use_a_single_password_for_everything.html
accessed 10 July 2009.

Wendland, Mike. 2008. "How secure are your text messages?" *Detroit Free Press* (Detroit, Michigan). January 24.