6.857 Final Project - One Time Pad

Julia Huang, Harini Kannan, Yuanqing (Kai) Xiao, Edwin Zhang

May 14, 2015

1 Problem

When Edward Snowden shook the world by releasing classified documents about the NSA's PRISM surveillance program, millions of Americans were forced to come to terms with a new reality. The NSA had complete access to Americans' Google and Yahoo accounts and had been searching through email and instant messaging records for years. More than ever, ordinary citizens realized the importance of secure messaging.

We wanted to create a simple yet completely secure messaging app for every day use, something that even the NSA could not break even if they had access to an entire database of encrypted messages. The one-time pad is the only method of encryption that is information-theoretically secure, or secure against an adversary with unlimited computing power. To encrypt a message using the one time pad, the plaintext is paired with a secret key of the same length (known as the pad), and then each byte i of the ciphertext is generated by xoring the ith byte of the plain text with the ith byte of the key. This technique is secure only if the key is truly random and no part of the key is reused.

When designing the app, we needed to address the following issues associated with using a one-time pad, despite its simplicity and security:

1. The need for true random one-time pad values
2. Difficulty in securely exchanging the pad

We will develop a messaging application that is able to both generate random values and easily and securely exchange the pad.

2 Background and Existing solutions

Many popular existing messaging apps have some serious security flaws. Both Facebook chat and Google Hangouts/chat do not encrypt messages such that the provider cannot read it - in other words, a third party can read anything one transmits over both Facebook and Google chat. Apple's FaceTime and iMessage both encrypt messages such that the provider itself cannot read it, making Apple's messaging products more secure than Facebook's and Google's. However, FaceTime, iMessage, Facebook chat, and Google Hangouts all do not verify the identity of contacts, meaning that messages could be forged.

According to the Electronic Frontier Foundation, there are currently only 6 messaging apps in the market that satisfy the following criteria:

1. Encrypted in transit
2. Encrypted so that the provider of the app cannot read it
3. Verifies the identities of contacts
4. Keeps past messages secure if keys are stolen
5. Opens up code to independent review
6. Properly documents security design
7. Has audited code recently

These six apps are ChatSecure, Cryptocat, RedPhone, Silent Phone, Silent Text, and TextSecure.

Zendo is a new messaging app that came out a couple months ago that uses a one-time pad to securely transmit messages. It is very similar to our app, requiring a physical pad exchange via a QR code. However, unlike Zendo, we plan to make our app open source and available for independent review by other developers.

3 Implementation and Approach

In order to create a messaging application that uses the one time pad, we need to:

- generate truly random data,
- securely transmit data, and
- keep track of what part of the pad is used for each message.

We will detail how we implemented each of these steps below.

3.1 Generating truly random data

We will use environmental data to generate the random bits that act as the pad between two people. Suppose that two people, Alice and Bob, would like to send encrypted messages to each other. Our app allows Alice to take a picture of her surroundings and will proceed to extract randomness from small variations in the individual pixels of the photos.

For our specific randomness generator, we extracted randomness from a photo through the following steps.

3.2 One random bit per pixel:

We map each individual pixel of the photo to a single bit of randomness. To do so, we begin by taking the RGB values of each pixel, and representing them as 8-bit binary numbers. We then concatenate the 3 values together to get a 24-bit binary number. If these bits are $b_1, b_2, \ldots, b_{24}$, then the random bit that we generate corresponding to this pixel is defined as $r = b_1 \oplus b_2 \oplus \cdots \oplus b_{24}$. For most photos of one's surroundings or scenery, this value should be random. To ensure randomness with properties that we want, we add in two special filters.

3.3 Filter 1: Pseudo-Random Filtering

Randomly decide whether or not use each bit of the photo with probability $\frac{1}{k}$, where the probability $\frac{1}{k}$ is just a pseudo-randomly generated event corresponding to a normal phone's random number generator and k is an integer that we choose after some testing. We found that $k = 40$ worked fairly well for our purposes. This is just to reduce the chance of clusters of extremely similar pixels generating long strings of the same bit value, which shouldn't happen often anyways, given how precise pixel RGB values are.

3.4 Filter 2: The von Neumann randomness extractor

At times, the pixels of the photos generate bits that appear mostly random, but appear biased one way or the other. In order to ensure that the random bits occur with equal frequency, we can use the von Neumann randomness extractor to change biased randomness to uniform randomness.

In our case, we begin with a long string of 0's and 1's. We first need to check that the appearance of 0's and 1's are independent. We note that they are independent if the probability that a 1 follows a 0 is the same as the probability that a 1 follows a 1, and if the probability that a 0 follows a 0 is the same as the probability that a 0 follows a 1. We can then see the distribution of substrings of length 2. We can use the chi-squared test to test for random data. Running our data through the chi-squared test, we receive a p-value of 0.21552397088219707. Since the p-value is pretty high and thus greater than our significance level, we do not reject the null hypothesis, and it is unlikely that our bits are not independent of the previous bit.

Now, we look at every consecutive pair of bits, not counting overlaps. That is, we look at bits 1 and 2, then bits 3 and 4, and so on. We then construct a new random string based on the value of each pair of consecutive pair of bits. If the consecutive bits are x and y, we define the generated bit in the new string as

$$f(x,y) = \begin{cases} 1 & : (x,y) = (1,0) \\ 0 & : (x,y) = (0,1) \\ \emptyset & \quad : \text{otherwise} \end{cases}$$

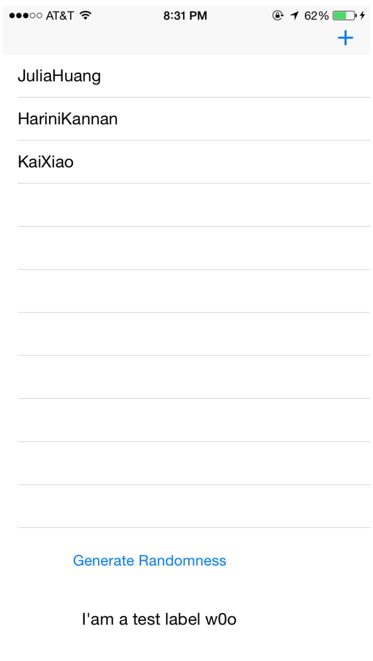
Here, $\emptyset$ simply means that we don't generate a bit if $x = y$. It can easily be shown that this converts biased randomness into unbiased randomness. To prove this result, we begin with a biased source of randomness that takes on the value 0 with probability p and the value 1 with probability $q = 1 - p$. Then the probability that we see a 1 in the new random string is qp and the probability that we see a 0 in the new random string is pq. Thus this process gives us our desired uniform randomness.

This randomness extraction works most effectively with a photo that is more or less homogeneous throughout. If photos and pixel coloring works as predicted, this should give us a truly random strings of 0's and 1's. However, we must make sure that the data passes a series of randomness test in order to ensure that the data satisfies the criteria for being truly random, and does not have certain bad properties that make it more susceptible to attacks. In order to test for true randomness, we ran several tests for randomness, including a chi squared test and the FIPS randomness tests.

3.5 Transmitting data

One important design consideration is to make sure that the pad is long enough. In order to send a large amount of data at one time, we will encode the data into a series of QR codes that will be made into a gif that the other person can record. We will designate special QR codes to represent the start and end of a sequence, similar to start and stop codons in DNA. Because the data is exchanged by having one phone record a video of the other phone, it won't be possible for an adversary who is listening in on communication channels between the two phones to capture the exchanged data. The biggest security risk, in this case, would be an adversary somehow also being able to record the video of the same gif.

3.6 The Messaging App



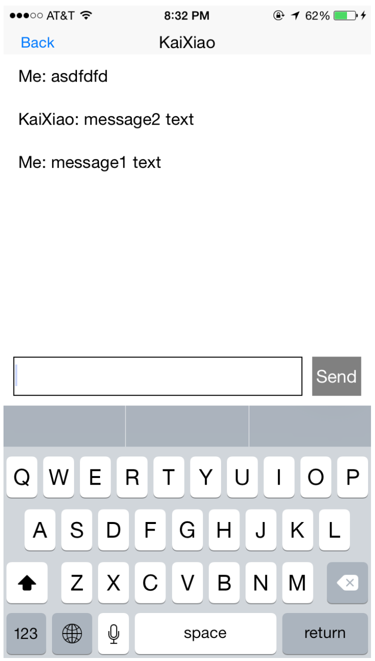
JuliaHuang

HariniKannan

KaiXiao

I'am a test label w0o

Figure 1: Main screen with a list of chats and the option to to generate a pad



Me: asdfdfd

KaiXiao: message2 text

Me: message1 text

Figure 2: Viewing messages between the user and another person

We created a messaging app to demonstrate our one time pad creation and encryption. To generate random bits, the user the camera phone to take a picture of their surrounding and using the von Neumann randomness extractor, we extract bits out of that picture to create our random pad. We then save the random pad to our phone, while the hash of the pad is stored on our database. Messages and conversations between the user and other people are loaded from our Firebase database and displayed on our main screen. All the encryption and decryption of the messages is done on the phone so even if our database is compromised, the messages are still secure.

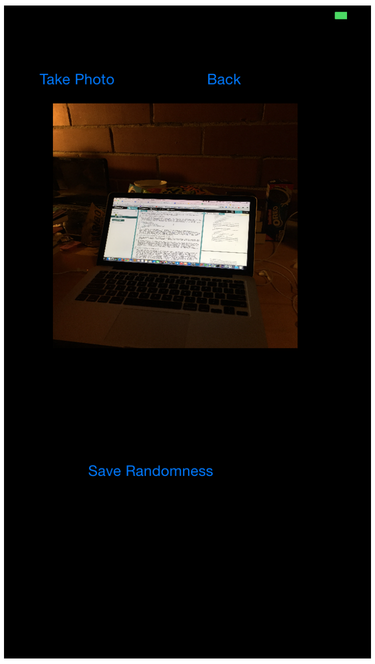


Figure 3: We can use the camera to take a picture to generate a random pad

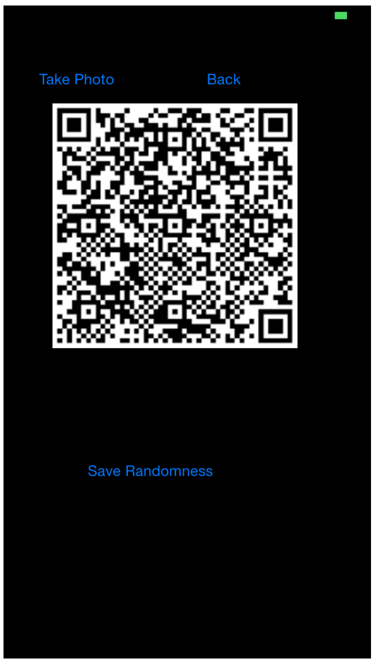


Figure 4: The generated QR code that is used to transfer the pad between two users

3.7 Firebase

To store messages, we created a database, powered by Firebase. The database stores the encrypted version of the message sent, as well as the hash of what pad was used to encrypt the data. The database also stores a list of the hashes of the pads that each user has access to so when sending a message, the app can figure out what pad to use to encrypt the message.

4 Security considerations

4.1 Avoiding reusing the pad

Reusing parts of the pad can lead to a large security leak. For example, if the adversary knows that the first part of the pad was used for encryption, he or she can simple XOR messages together to get the pad back. Even if certain bits were reused in the pad, a crib dragging technique and other forms of educated guessing with brute force dictionary attacks could ultimately crack the pad and subsequent messages. As a result, we will also need to keep track of which bits in the pad have been used to avoid reusing bits. To ensure that messages being sent at the same time in opposite directions, i.e. a message from Alice to Bob and a message from Bob to Alice sent at the same time, avoid using the same bits, we force the odd bits to be used when encrypting data from Alice to Bob, and the even bits of the pad to be used to pad the data from Bob to Alice. To expand to all users, if we have a messaging group of n people, the xth person when sorted alphabetically, or by some other predetermined method, will use the bits that are congruent to x mod n. This ensures that there is no overlap in bit usage, and thus, we have a secure encryption method.

4.2 Electronic Frontier Foundation's security conditions

We will now analyze how our app performs under the EFF's security conditions:

1. Encrypted in transit - YES

   Our app does indeed encrypt messages using the one-time pad as they are transmitted from the sender to receiver.

2. Encrypted so that the provider of the app cannot read it - YES

   All messages are encrypted and then stored in our database, powered by Firebase.

3. Verifies the identities of contacts - YES

   Because sending a message to a contact requires physically meeting up and exchanging the pad, a message from a contact cannot be forged by an adversary (as the adversary would need access to that contact's pad).

   While not within the scope of our final project, we hope to keep developing the app and add a one-time MAC authentication system to also verify the identities of our contacts.

4. Keeps past messages secure if keys are stolen - NO, but should not be a consideration

   If an adversary were to be able to steal the keys, he or she would be able to decrypt the messages easily. However, there is no way to steal keys from our central database because we do not store them there. As such, the only way to steal the keys would be to break into someone's iPhone. However, this would expose the messages anyway, as the messages are viewed on the phone in plaintext.

5. Opens up code to independent review - YES

   We plan to open-source the app.

6. Properly documents security design - YES

   We plan to document our security design if we decide to release the app.

7. Has audited code recently - YES

   We plan to open-source the app and invite other developers to audit its security.

5 Results

We used several chi-squared tests to test for uniform distribution as preliminary tests for randomness. We took the original stream of bits and split it up into non-overlapping substrings of various lengths and checked to see if the occurrences

Length	p-value
1 bit	0.87706545180540929
2 bits	0.82240313554599331
3 bits	0.88696348682261894
4 bits	0.95571930416694673

Table 1: chi square tests

of these substrings were roughly uniform. We had very promising high p values that indicated it was unlikely for the data to be not random.
We proceeded to use some FIPS 140-2 random number tests to test our generated data. Federal Information Processing Standards 140-2 publication for cryptographic modules details four statistical tests for randomness. The four tests are: the monobit test, the poker test, the runs test, and the long run test. Explicit bounds are provided for each test that the computed results must satisfy, and if any of the tests fail, then our randomness generator fails the FIPS 104-2 statistical test for randomness. The data we tested with passed all four tests, so it is quite likely for our random generator to be truly random.

5.1 The Monobit Test

The number of ones counted in a bit stream of 20,000 bits should be within the accepted region of 9,725 to 10,275 ones. The bit stream we have has a total of 9937 ones, thus passing the test.

5.2 The Poker Test

The bit stream is divided into 5000 non-overlapping consecutive 4-bit segments. We use a chi-squared test, and expect that for each of the 16 types of segments to have roughly the same frequency. We thus expect the statistic $\frac{16}{5000} \cdot \Sigma_{i=0}^{15} g_i^2 - 5000$ to be between 2.16 and 46.17. From our stream, we calculate our statistic to be 19, and we pass the test.

5.3 The Runs Test

We want to ensure that if we split it into substrings with consecutive values of one or zero, the frequencies of these substrings of different lengths are reasonable. We expect and have results: t Since the number of runs of each length land in

Length of runs	acceptance	runs for 0s	runs for 1s
1 bit	2315 <= x <= 2685	2461	2541
2 bits	1114 <= x <= 1386	1257	1205
3 bits	527 <= x <= 723	635	637
4 bits	240 <= x <= 384	322	305
5 bits	103 <= x <= 209	174	138
6 bits	103 <= x <= 209	145	168

Table 2: frequency of runs

the acceptance region, we pass the test.

5.4 The Long Run Test

We also don't want any run to be too long. It is unreasonable to see a run of length 26 or greater. The length of the longest run using our 20000 bit stream is 14, and we thus pass the test.

5.5 Final Remarks

Since all of the FIPS tests are passed, we have good reason to believe that the data we generated is random, and our app is thus secure. We made a proof of concept app that allows people to securely message others using the concept of the one time pad. We generated randomness via collapsing all the RGB byte values into one bit, and then proceeded to extract randomness using the von Neumann extractor. We then exchanged the pad by having one person take a video of the other person's QR code gif, and this allowed us to exchange the pad without a server in order to encrypt and decrypt the data securely. Thus, we have created an app with secure messaging. You can find the implementation of our app here: https://github.mit.edu/julia-h/6.857.git.

6 Bibliography

References

[1] Lomas, Natasha. "Zendo Is My New Favorite Secure Messaging App." TechCrunch. N.p., 24 Mar. 2015. Web. 11 May 2015.

[2] Mlot, Stephanie. "Only 6 Messaging Apps Are Truly Secure." *PCMAG*. Ziff Davis, LLC, 5 Nov. 2014. Web. 13 May 2015.

[3] "Secure Messaging Scorecard." *Electronic Frontier Foundation*. N.p., 03 Nov. 2014. Web. 11 May 2015.

[4] Vithanage, Ananda, and Takakuni Shimizu. "Fips 140-2 Random Number Tests." (2011): n. pag. FDK Corporation, 2003 Feb. 10. Web. 11 May 2015.