Crypto-Juice: One-time Pad Confidentiality for TCP Applications

Blake Howard and Steven Miller
Department of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210
{howard.1003, miller.5252}@osu.edu

1. Introduction

Modern encryption products intend to protect against compromise by selecting a key size such that the most powerful cryptanalysis platforms cannot crack the ciphertext within the foreseeable future. Communication through such products, however, may be compromised by future, powerful machines if ciphertext transmissions have been stored. The only encryption scheme to date believed to be mathematically secure against an adversary with unlimited computational power is the one-time pad. We propose to implement the one-time pad scheme using a large, pre-shared key. Such a system could be useful to for protecting highly sensitive communications, but only be able to transmit a limited amount of data. In such a system, we call the remaining random data in the pre-shared key "crypto-juice."

2. Related Works

Two fundamental tools in network security and privacy and openSSL and VPNs. OpenSSL allows user to use the open source libraries used to implement the SSL and TLS protocols along with an array of other cryptographic tools such as certificate management, hashing and encryption algorithms, pseudorandom byte generators, timestamping tools, and many others. VPNs allow users to extend private networks to be used across public ones. By doing this, a user can better stay both anonymous and secure. We have taken ideas from both of these resources to implement crypto juice. From the openSSL side, we have tested our program by using SSH. The concept of a VPN is a big influence on our final project because crypto-juice could in theory be used as a security boost or additional feature to an already implemented VPN.

3. Background

As mentioned, the foundation of our security scheme is based on a one-time pad structure. The idea behind a one-time pad is using a random key that is as long as the message being encrypted so the key cannot be repeated or recovered. The key is used to encrypt the message and then subsequently decrypt it after it has been transmitted. Our implementation expands on this idea and uses a large amount of pre-shared random data as key and encrypts and decrypts messages using parts of this key. The strength of one-time pad comes from the random data that bears no statistical relationship to the plaintext. Because of this, the encryption scheme is unbreakable.

Despite one-time pad's substantial security strength, it has often been regarded as an impractical tool due to its necessity of large amounts of random data given to both communicating parties. This being said, one-time pad is a viable solution for this product because initially, our ideal user would use crypto-juice to establish private communication between a home office and their work. Under this assumption, the user could create the large amount of random data at work, keep it on a hard drive, and take to home in order to establish a secure connection. At this point, one-time pad would not be ideal for excessively long distance communication or for transmitting excessively large amounts of data.

4. System Design

The system was designed as a one-time pad, assuming a large pre-shared key. It was intended to be an added layer of confidentiality for TCP applications.

The large, pre-shared key storage was designed with a modular approach. Any file may be specified as the random data key. The random data generation is thusly independent of the network application. Random data generation may be changed as requirements change. Only truly random data will suffice to protect against an adversary with unlimited computational power.

The network application was designed to enable a client to connect to a remote TCP application through a process on his or her local machine by using port forwarding. A server-side process connects to the service and provides another service of its own. A client-side process would then connect to that process. Then, a client may connect to the client-side process to access the remote service. The server and client side processes are in place to modify the data stream by encrypting and decrypting the data with XOR operations between the application data and the random data.

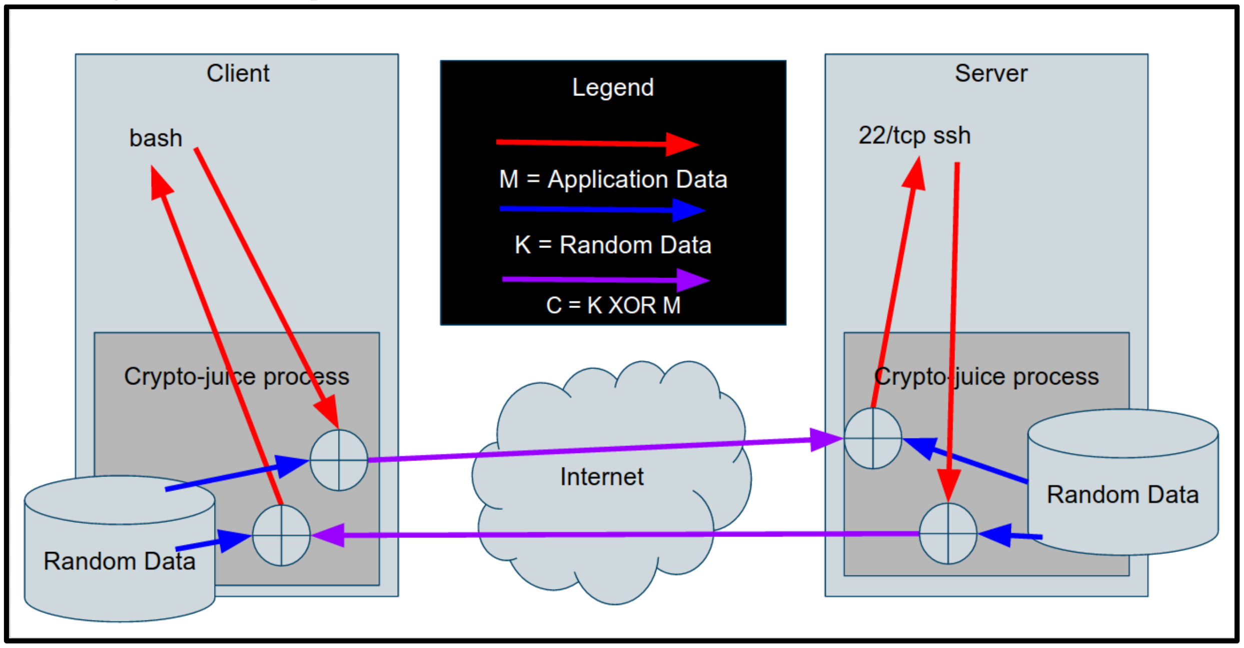


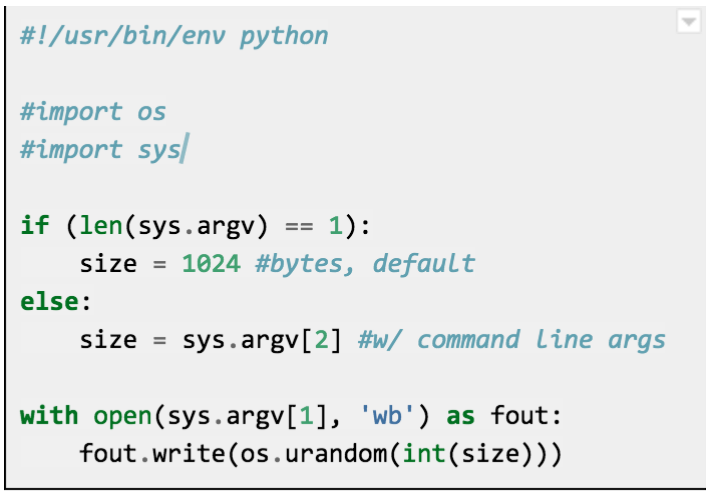
Figure 4.1: System Design

5. Implementation

5.1 Random Number Generation

One of the most important parts of this project was discovering and deciding on an appropriate method for random number generation. For the most secure system possible, the best choice for crypto-juice would be an implementation using true random number generation. With TRNG, numbers are generated based on analog factors using a physical process as opposed to using a strictly software approach. Through this strategy, prediction of the future random data based on previous pieces would be highly improbable.

Due to time and resource restraints, for the sake of expediting the implementation of crypto-juice, we decided to use a software approach to creating the random data. In our process, we explored two possible methods using two language classes. The first class used was Java's SecureRandom class. SecureRandom provides cryptographically strong random numbers which comply with standards according to FIPS Security Requirements for Cryptographic Modules. Using Java's tools would create a cryptographically secure portion of random data, however due to the rest of this project being implemented in Python, we decided to explore what options offered in the field of PRNG. Python's os library offers a secure RNG that does not rely on software state and sequences and are not reproducible. After this python implementation was discovered, we changed from using Java's SecureRandom to Python's OS urandom option.



```
#!/usr/bin/env python

#import os
#import sys

if (len(sys.argv) == 1):
    size = 1024 #bytes, default
else:
    size = sys.argv[2] #w/ command line args

with open(sys.argv[1], 'wb') as fout:
    fout.write(os.urandom(int(size)))
```

Figure 5.1.1: genRanBytes implementation

Figure 5.1.1 above shows our implementation of the genRanBytes python function, which, given command line arguments of output file and size in bytes, will produce a file in the user's working directory of however many random bytes as specified. From here, the user can then subsequently invoke the main program, establishing a secure connection using the previously generated random data.

5.2 Network Protocol and Connection

The network application was designed such that the client may access the remote service from a local process. The port forwarding was implemented and tested in Python 2.6.6. The server-side crypto-juice process makes a tcp connect call to the local service to be forwarded (ssh as seen in Figure 4.1). After the connection is established, the process binds to a port specified as a command line argument and waits to accept a connection from a remote machine. The client-side crypto-juice process then makes a tcp-connect call to the port opened server side. After the connection is established, the client-side crypto-juice process waits to accept connections from the local host. Thus, a channel is created from the server to the client. The client and server side crypto-juice processes move data from the transport layer back to the application layer before forwarding in order to implement the extra application-layer security. This step is further explained in section 5.3.

In the original implementation, communication between the client and server crypto-juice process contained the encrypted data, prefixed by a header containing the ciphertext length and index from which to decrypt in the random data file. It was determined that such an implementation is insecure to man in the middle attacks. Relying on decryption indices in a plaintext header may lead to potential exploits where a piece of the key file may incorrectly be used for encryption more than once. In the revised version, no header was included. The transmitted data was only the application data encrypted with one-time pad. Decryption index in the random data is determined by the total amount of data that has been received as described in the following section.

5.3 Encryption and Decryption

The main idea behind crypto-juice's organization when it comes to using the random data that has been generated is that the client side will encrypt and decrypt using data from the front of the file, while the server side will encrypt and decrypt using the data from the back. Once any part of the data from either side is used to encrypt one time, that data is no longer able to be used to encrypt again, hence one-time pad. The available and used data is kept track of using indices in the encryption and decryption functions. Figures 5.3.1 - 5.3.4 below show an abstract example of how the encryption scheme work. For a message from client side to server side, the message is brought into the program and a piece of the random file the size of the message is taken from the front of the file is allotted to encrypt that message. The message is then XORed with the random data and it is transmitted. Once on the server side, the message is then decrypted using its size and an index of what data it used to be encrypted. Similarly for a message from server side to client side, the message is introduced in the same fashion as before, but now the random data is taken from the back of the file. The message is then transmitted and decrypted analogously to client to server described before. This separation is important because it eliminates the possibility accidentally reusing random data for encryption.

Figure 5.3.5 visualizes what happens when there is not a sufficient amount of crypto-juice for encrypting the message. When a message comes in, its size is compared to the index of the remaining random data. This includes the index from the back of the file and the front. If there is not enough data to encrypt the message, the program terminates and the last message is not sent.

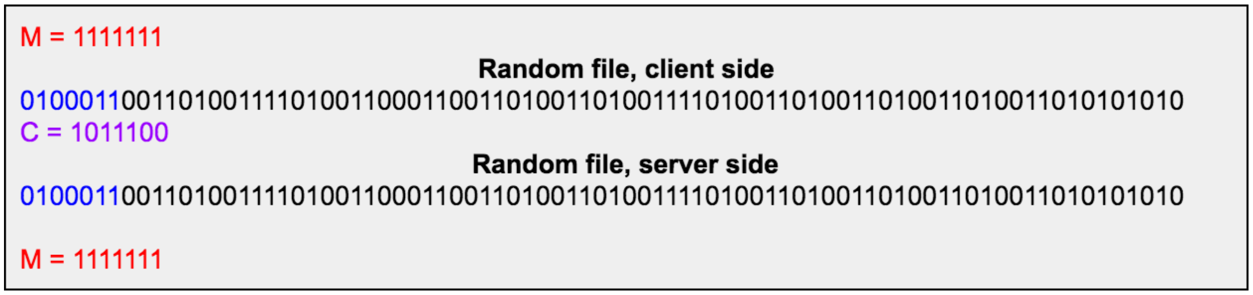


Figure 5.3.1: Encryption Process, client to server example 1

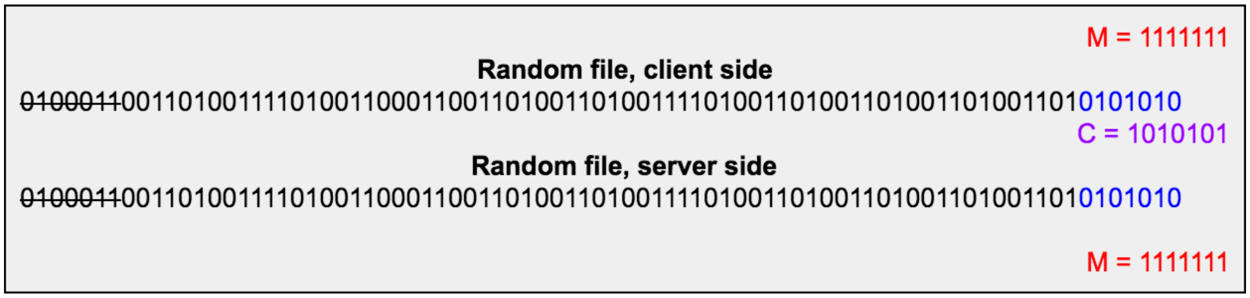


Figure 5.3.2: Encryption Process, server to client example 1

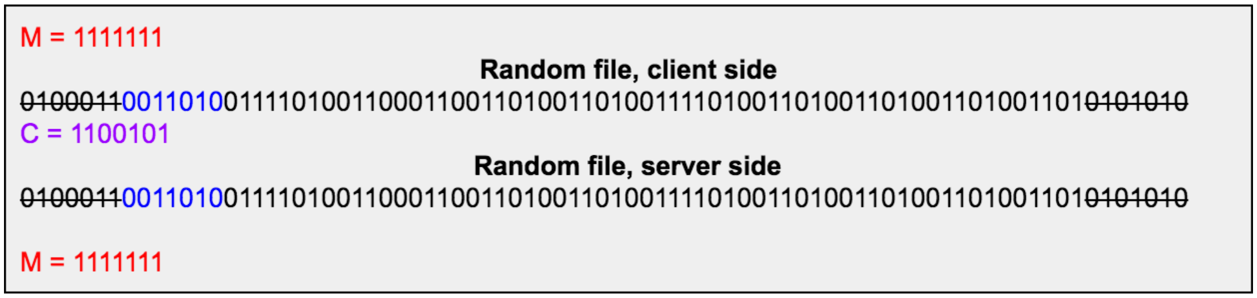


Figure 5.3.3: Encryption Process, client to server example 2

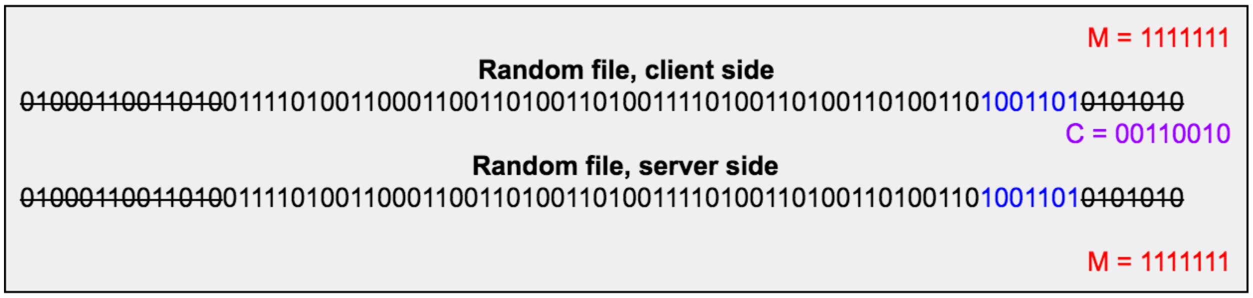


Figure 5.3.4: Encryption Process, server to client example 2

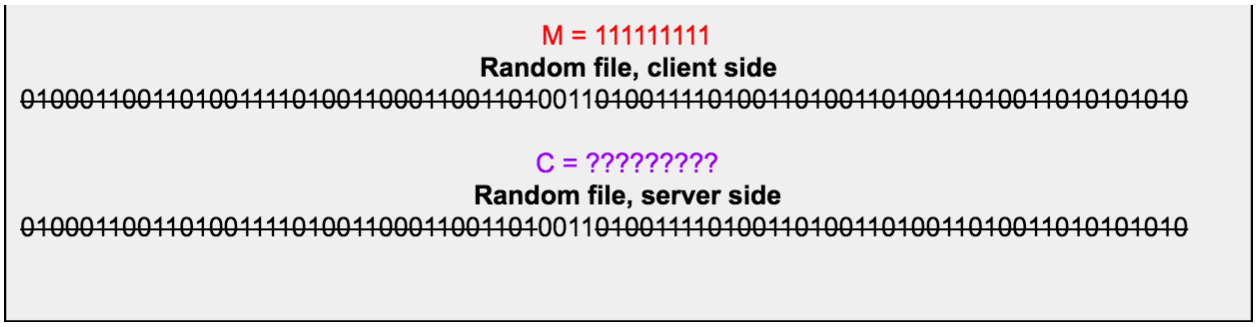
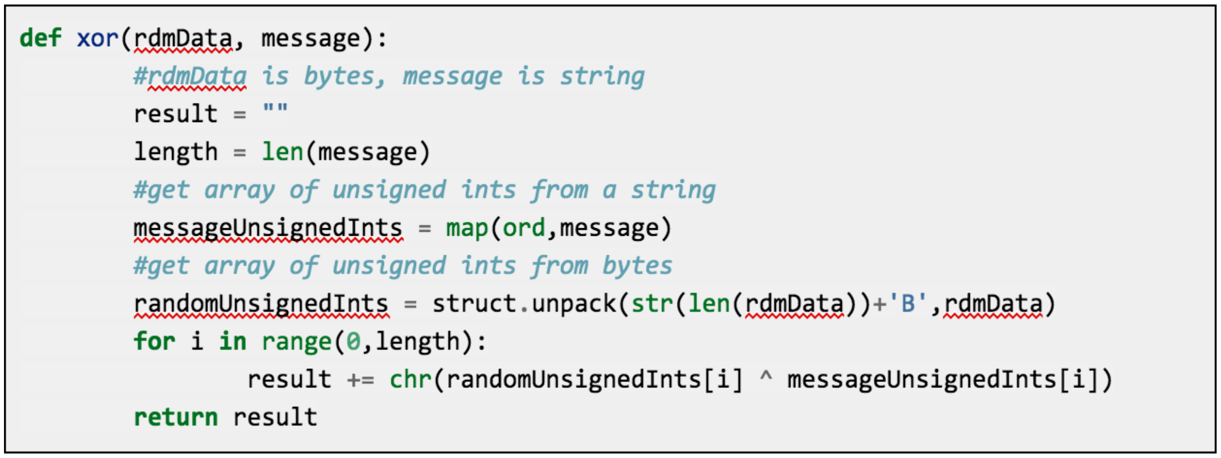


Figure 5.3.5: Encryption Process, client to server insufficient random data example

The encrypting and decrypting XOR function was implemented and tested in Python 2.6.6. The bitwise XOR operation is implemented only for unsigned integers in Python 2.6.6. Data was read from the random file into the Python bytes type. Message data from the application data stream was received into a Python string type. The XOR function was implemented to bitwise XOR two parameters of equal size, one of type bytes and one of type string. The string was converted to an array of 8 bit unsigned integers by using the map() function. The bytes type was converted to an array of 8 bit unsigned integers by using the struct.unpack() function. The two 8 bit unsigned integer arrays were then iterated through, XORing each element of the two arrays by matching index. For each XOR, the resultant 8 bit integer was converted into a character and appended to the ciphertext.



```
def xor(rdmData, message):
        #rdmData is bytes, message is string
        result = ""
        length = len(message)
        #get array of unsigned ints from a string
        messageUnsignedInts = map(ord,message)
        #get array of unsigned ints from bytes
        randomUnsignedInts = struct.unpack(str(len(rdmData))+'B',rdmData)
        for i in range(0,length):
                result += chr(randomUnsignedInts[i] ^ messageUnsignedInts[i])
        return result
```

Figure 5.3.6: XOR implementation

6. Evaluation & Discussion

6.1 Key file

As mentioned, the key file is a crucial element in the use of crypto-juice. Because of this, any limitations on it are inherently limitation on the whole system. We were able to successfully implement a program that generates an arbitrarily large amount of random data, albeit pseudorandom. Because of its cryptographically secure characteristics, having the data be pseudorandom was sufficient for crypto-juice up to this point, but if it were going to be put into production or used on a larger, more fragile scale, implementing a version of the random number generator using true randomness is a necessity.

Another limitation that was the event of a system compromise which would lead to an adversary having access to the key file. At this point in time, if this were to happen, the data would remain on the disk. One solution to this would be adopting an "ephemeral" key strategy. With this, if a system is compromised, the key will no longer be stored on the disk.

6.2 Network Protocol

The application was successfully implemented to XOR binary data from a file with streaming application data to provide confidentiality. The application should be compatible with any TCP client/server application. The system was tested to work successfully with netcat and secure shell (ssh).

The application was designed to be an extra layer of confidentiality. It is important to note that the system is vulnerable to attacks to compromise the data integrity. TCP is relied on for message integrity, but such practice is not cryptographically secure. Modified messages should be noted as invalid by TCP and automatically retransmitted, but the only protection is a fast 16 bit checksum. An adversary performing a man in the middle attack may craft a modified message to match the checksum to corrupt the data for that message, but not to learn the contents.

Another potential vulnerability arises from the lack of authentication. When the server side crypto-juice process is expecting a remote connection, any client may connect to the service. A malicious client connecting to the service cannot decipher the data, but can deny availability. Also the malicious client can send random data to the service and the server side process will think that the client is sending garbage data. The possibility for a maintained connection from an adversary is an attack vector.

It is advised to use the crypto-juice application on top of other applications that provide authenticity and integrity, such as ssh. Since ssh is designed to defend against man in the middle attacks, and the crypto-juice processes are acting "in the middle", tunneling through the crypto-juice processes should not decrease the security of ssh.

7. Conclusion

A one-time pad scheme is undoubtedly secure, but traditionally difficult or illogical to apply to most situations. Crypto-juice provides a one-time pad solution to be used to create a secure TCP tunnel for two parties to communicate safely. To achieve this secure connection, a few things need to happen. Firstly, a large amount of random data needs to be generated in order to have reliable encryption. This data was generated using python's os library through OS.urandom, a cryptographically secure pseudorandom generator solution. Next, once the data is generated the connection can be made by the server to the client. From here messages from client-to-server are encrypted and decrypted from data at the front of the key file, while server-to-client messages are encrypted and decrypted from data at the end. Once the data in the key file has ran out, the one-time is no longer possible and the connection terminates.

This secure tunnel is ideal for users who need to transmit small amounts of highly critical data over the internet. Such a connection may be made with a company server and an employee working remotely. The current state of crypto-juice is working well, but there is room for expansion of functionality and improvements in security as listed above. Once these improvements are made and the system is hardened further, it is possible that crypto-juice could play a role in protecting others' sensitive information.

7. Acknowledgements

- Blake Howard
  - Implemented XOR
  - Implemented random data generation
  - Implemented algorithm to extract data from key file
- Steven Miller
  - System Design
  - Implemented network connection and protocol
  - Refined XOR