Cryptography Final Project

By: Owen Lockwood, Samarth Patel and Kousei Arima

White Hat

FOR BLACKHAT TEAM/TAs: The README is in the other file. Note too for blackhat (and grading TAs) that the other file contains information on key exchange and handshaking.

This section outlines the white hat portion of the project, i.e. the development and implementation of a ATM/Bank secure communication. Let us first note that this document does not contain instructions for running the code (that is outlined in the README), rather it outlines what our code does, what algorithms we used and implemented, and our rationale behind these decisions.

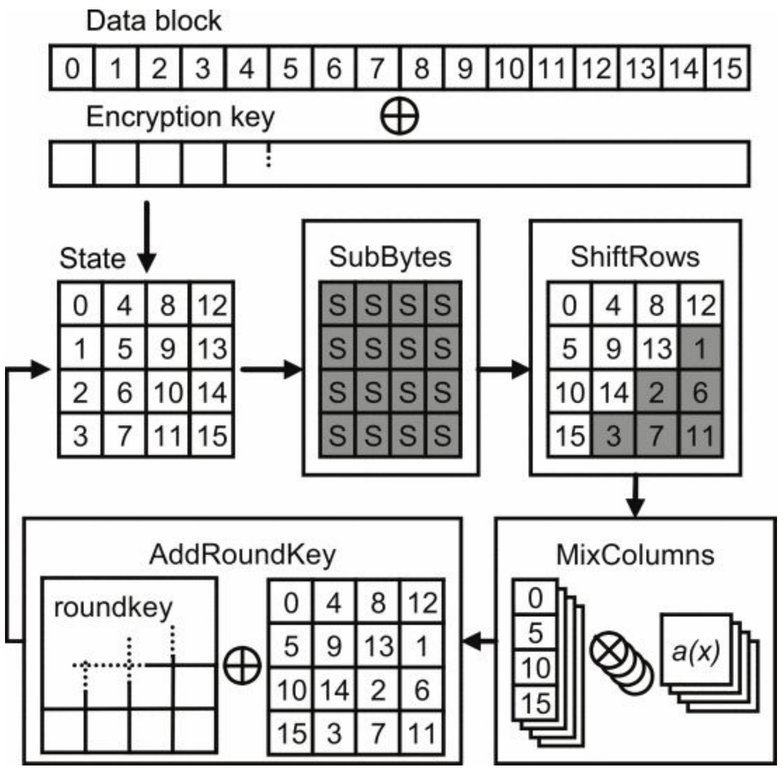
Before beginning to design an ATM/Bank system, the first step is to identify and model threats. The design of a cryptosystem for non-internet connected, physically inaccessible, security through obsolescence computers within the pentagon is very different from modeling a cryptosystem of cloud storage for users around the world. Hence, it is critical to identify security objectives and investigate how these objectives could be compromised. Here we have already implicitly defined our threat model as an adversarial one, one in which we are focused on possible attacks and preventing them before they come to fruition. First, let us outline exactly what the overarching idea of this project is (as there may be slight differences in interpretations of the instructions). We envisage this system as a two party one. Only a given ATM should be allowed to access the bank and the only thing the ATM should access is the specified bank. The goal is to establish a secure connection between the ATM and Bank that cannot be intercepted and decrypted or interfered with. To this end we build our cryptosystem out of several different components, each serving a different role in the protection of information and the maintenance of security. We use a symmetric encryption scheme to communicate between the bank and the ATM in order to make sure no outsiders can intercept and understand the messages. We use a Message Authentication Code (MAC), specifically one based on a hash function (HMAC) to ensure that might be intercepted on route and modified are not interpreted by either party as authentic. We establish this connection via a handshake protocol and a public key exchange. Finally, to ensure that the ATM is not being misled, we implement a digital signature algorithm so the bank signs every message it sends.

In the following sections we will outline exactly which algorithms we implemented, how we implemented them (including any variations or changes) and, importantly, the reasoning behind our algorithm decisions.

Symmetric Encryption

The symmetric encryption we implemented was Advanced Encryption Standard [1] with a 256 bit key size (AES-256), the largest of the 3 commonly used key sizes (128, 192, 256). Before we get into the algorithmic details of AES-256, it is important to understand the reasoning behind this decision. AES-256 is significantly more secure than DES, and represents a significant challenge to crack. Recently a related-key technique for breaking 10 round AES-256 in 2^{45} time was presented [2]; however, on full AES-256 (14 rounds) and only a single key, current works suggest cracking time complexity of $2^{254.27}$ [3]. Not only are these attacks largely computationally infeasible, they are especially difficult due to the nature of our cryptosystem. The ATM and Bank generate a different key every connection and connections would likely last a matter of minutes at most. Because there is no key reuse and because the session is relatively short, AES-256 significantly reduces any threats there may be for a 3rd party to decode and read the messages between the Bank and ATM for any given session. AES-256 is also sufficiently fast that there are no usability concerns. Because Bank, ATM communication is very private, it is essential to maintain the secrecy of the messages, hence why we chose one of the most prominent symmetric encryption standards and why we used the largest key size.

Now we will give a general outline of the actual algorithm as there are some sizable differences between AES-256 and the SDES we implemented in class. We will ignore small implementation details (like ascii conversion and the like) to just go over the steps of the algorithm. The first step is the key expansion. The input is a 256 bit (32 byte) starting key. This is expanded into 15 keys for AES-256 as there is one initial key XOR and 14 rounds. In order to do this key expansion there are basically 3 repeated steps that are applied to generate new keys, a key rotation, an exponentiation in GF(2^8) and the using Rijndael's Sbox (which is simply a known box, that is hard coded). With key expansion done, the AES Rounds begin. Note that AES works on a 4x4 matrix of information. First there is an initial XOR with the key and message, then the 14 rounds begin. A general outline of the rounds can be seen to the left, in a diagram from [4]. The first step is to substitute bytes. The bytes in each index of the matrix are



substituted with the sbox of that index. Next is the shift rows operation, which has no effect on the 0th row, but left rotates the other rows by their row number (i.e. rotates row 1 by 1, etc.). The next step is mixing columns. This can be shown succinctly by the matrix multiplication to the left, done within GF(2^8). These 4 steps are repeated 13 times in AES-256. For the last round, only byte substitution, row shifting and key XORing is done.

$$\begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} \begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{pmatrix}$$

Message Authentication Code

The MAC is done using a hashing function, Secure Hash Algorithm 1 (SHA1). We chose SHA1 because we determined that the potential attacks against it are minimal given the conceptualization of our cryptosystem. Although many organizations are shifting to more complex and larger hash functions, we believe SHA1 to be sufficient for our needs. There have been a number of recent high profile attacks and techniques for breaking SHA1, so it is important to address these. [6] presents a chosen prefix attack with 2^{67} complexity, [7] presents a collision with 2^{63} complexity and [5] implements a real attack with comparable complexities that took 2 months on 900 GPUs (~$75,000). Given our threat model, we believe this demonstrates that SHA1 is sufficient. Because this is solely ATM/Bank communication, the maximum withdrawal amount set forth by the Bank Secrecy Act of 1970 is $10,000 (above which banks are required to report and most will not let you withdraw). In addition, most banks have stricter ATM withdrawal amount restrictions. If you have $75,000 to buy GPUs and months of time (for just 1 small part of the whole cryptosystem), the amount of money you would have access to is significantly less than what you already have (in addition, just cracking SHA1 would be far from complete access to funds). And we therefore consider that threat to be minimal.

Now to go over the actual SHA1 algorithm. This algorithm is pretty straightforward (another incentive for using it) and we provide a general overview (the exact code may slightly differ here because we do most of the algorithm with binary rather than hex as is often done). First, the message is split into 512 bit blocks. For each block the following steps are done. The block is broken into 16 32 bit subblocks. These are then extended via rotations and XORs to be 80 32 bit subblocks. Then for each of these 80 subblocks a boolean function is done with starting values: a = 0x67452301, b = 0xEFCDAB89, c = 0x98BADCFE, d = 0x10325476, e = 0xC3D2E1F0. Depending on the subblock, there is a different function. At the end these values are modified and swapped around. After all blocks are done, the initial values (aforementioned) are added with the current values (after all the blocks/functions) and the final result is the 160 bit value produced by concatenating these values. Because it is a hash function, it will reduce any size message to this 160 bit final product.

Creating a HMAC once you have the hash function is a relatively trivial matter. All that is needed is to generate the hash from the message, append it to the message, encrypt the message, and send it. When the receiver gets the message, they decrypt and hash what they decrypted. If this hash matches the hash that the sender sent, the message is accepted, otherwise it is rejected. This ensures that if the message is intercepted and modified between the two parties, neither would interpret it as a valid message.

Digital Signature

We use the Digital Signature Algorithm (DSA). DSA is a common algorithm for digital signatures and we use it because of its well established history and because of the nature of 'certificate authorities' allows us to assume a certain level of trustworthiness. The DSA algorithm takes three parameters: (p, q, g), where p and q are 1024 and 160 bit primes, respectively and $g = h^{\{p-1\}/q} mod\, p$. 1024 and 160 are a standard length of these two primes. P should be a multiple of 64 and q is the same length as the hash function output we are using, In addition, there is the private/public key pair, x, y that is used for verification. SHA1. In order to sign a message a random integer, i, is chosen, with a maximum of q-1. We then set r = g^i mod p mod q and s = (i^{-1}(SHA1(message) + x*r)) mod q. The signature for a message is therefore (r,s). To verify a signature, we first make sure r and s are in the appropriate ranges (maximum of q-1). Compute w = s^{-1} mod q, u_1 = SHA1(message)*w mod q and u_2 = r*w mod q, and v = $g^{u1}y^{u2}$ mod p mod q. If v == r, accept the message, else reject it.

This is conceptually straightforward. In general, you generate a private/public key pair and encrypt a message with your private key. Whoever you send it two can decrypt it with your public key and if the message is correct, verify one's identity. This is because only the authentic individual should have access to the private key, thus you can ensure that this message came from them.

In reality, these input values would be controlled by a trusted 3rd party authority. In order to simulate this concept we generated these values (generating random primes of 1024 bits and 160 bits using Miller-Rabin's primality test) and saving these to a 'keys' file that serves as a trusted authority or private document.

Handshake/PKC

Our handshake is a Diffie Hellman key exchange to determine the shared key used for AES during the session. The client generates random a, and sends server G^a. Server generates random b and responds with G^b. Client calculates (G^b)^a mod N and server calculates (G^a)^b mod N, which is the shared key. Because we only implement one protocol in our server/client (AES), there is no need for the client and server to communicate any additional info. We choose DHE because it provides forward secrecy: even if the key exchange is decrypted in the future, it does not compromise the session key because the key is never sent by either party.

References

[1] Daemen, J., & Rijmen, V. (2001). Announcing the advanced encryption standard (aes). *Federal Information Processing Standards Publication*, *197*.
[2] Biryukov, A., Dunkelman, O., Keller, N., Khovratovich, D., & Shamir, A. (2010, May). Key recovery attacks of practical complexity on AES-256 variants with up to 10 rounds. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (pp. 299-319). Springer, Berlin, Heidelberg.
[3] Tao, B., & Wu, H. (2015, June). Improving the biclique cryptanalysis of AES. In *Australasian Conference on Information Security and Privacy* (pp. 39-56). Springer, Cham.
[4] Hamalainen, P., Alho, T., Hannikainen, M., & Hamalainen, T. D. (2006, August). Design and implementation of low-area and low-power AES encryption hardware core. In *9th EUROMICRO conference on digital system design (DSD'06)* (pp. 577-583). IEEE.
[5] Leurent, G., & Peyrin, T. SHA-1 is a Shambles.
[6] Joux, A., & Peyrin, T. (2007, August). Hash functions and the (amplified) boomerang attack. In *Annual International Cryptology Conference* (pp. 244-263). Springer, Berlin, Heidelberg.
[7] Stevens, M., Bursztein, E., Karpman, P., Albertini, A., & Markov, Y. (2017, August). The first collision for full SHA-1. In *Annual International Cryptology Conference* (pp. 570-596). Springer, Cham.