The current weakness of the SDES implementation includes a small block size of 10 bits along with a key size of 10 bits. This means that there are only 2^8 = 256 possible original plain text with 2^10= 1024 possible keys. This means that an exhaustive search attack is possible where an attacker could simply try all the keys until one of them works and given the current computational power given to all modern computers, that's not going to take very long. This also means that all sorts of Brute-force attacks are possible and impossible for this data to be secure. Another weakness of the implementation is that the key is hardcoded into the encryption, meaning that once the key is found out, all other messages have been decoded by the attacker who has figured out the key. The same goes for the original plaintext. If an attacker gains both key (1100011110) and original plaintext (00101000), all messages past, present, and future , compromising confidentiality and integrity of the CIA triad (Confidentiality, Integrity, Availability) which is important for security. Another weakness of the implementation is hardcoded boxes substitution boxes in the round function, which wouldn't help being found out by cryptanalysis techniques such as linear and differential cryptanalysis. Linear cryptanalysis works by finding linear approximations between keys, plaintext, and ciphertext. It then builds an equation that uses all the related information that they can gather. By building a number of plaintext-ciphertext pairs, they can deduce the key. Differential Cryptanalysis works by focusing on the behavior of the cipher and then uses the differences in plaintext and ciphertext to find out the key being used. These are also known as analytic attacks. Another weakness of the implementation is timing attacks. Timing attacks utilize the timing required to perform each part of the cryptographic algorithm to predict the internal state of the cryptographic algorithm, for example, the time it takes for SDES to do the round function might clue in to the attacker that the SDES is doing the round function 5 times and thus much close to the full internal state of the algorithm. Times to process the algorithm and utilize the key to encrypt the algorithm can also

part of the timing attack as they can tell how many bits are utilized in the key and how long the input might be, or at least narrow it down so that they can use brute force attacks to find them.

The current weakness of the Triple SDES is like that of SDES although some have been eliminated. I will now talk about the weaknesses of Triples SDES

The keys have been hardcoded into the function, meaning that the key has no rotation, new key generation, or secure storage. This also means that once a person gains access to the three keys used for Triple SDES, then that means all communications that the encryption has been used on have been decrypted and looked at by third parties. Another weakness of the Triple SDES algorithm is that like the original SDES algorithm, it is limited by the small key size, which is 10 bits. And since it's a triple SDES, that means the key size is 30 bits. This makes it a little bit tougher than the SDES, but still not impossible to brute force like using an exhaustive search attack. The implementation of the Triple SDES attack also have similar weakness to SDES since the implementation uses SDES code as well. Timing attacks also work on Triple SDES, as timing the encryption time would reveal what kind of encryption has happened. The implementation's decryption and encryption only has the message being encrypted or decrypted 3 times in a row utilizing the keys created by the extract_3_keys, meaning that if the person has the full 30 bit key, encrypting 3 times with the 3 subkeys would not help. While Triple SDES is generally more secure than secure SDES since it is literally more encryption since it's done 3 times, it is still not immune to Cryptanalysis and techniques that worked on SDES such as linear cryptanalysis and differential cryptanalysis.

For Paillier, it is harder to find a weakness in the implementation of it. The key generation of this implementation have incredibly high numbers, being (466920160910299067185 3, 38775788043632640001, 1234567892, 9876543212,

791534681801253854346031374961854045149948519), making it incredibly hard to find the encryption in encrypt_all or find the private key. Paillier utilizes p,q, and n in order to generate the private key and the last 3 numbers in the group above are very large. However, this may make a weakness in the implementation itself, even if it's currently impossible with our computational power (except maybe the quantum computer). Brute force is a tactic that may work on weaker Paillier Cryptosystems, like trying to find the n of the cryptosystem, being the modulus of p and q. If you find n, you can more easily find p and q. However, since n, p, and q are all very large, this might take a lot more time and patience to find, making it impractical here. Factorization of the N modulus into its p and q are also hard. Even if you can find N, you would still have to find p and q and factorization will computationally difficult even with certain limitations like the numbers being semi-prime. A timing attack would work against this Paillier cryptosystem. Timing the cryptographic operations along with monitoring power consumption could clue in on how long calculations were for the encryption and therefore come up with estimations on N modulus of p and q. Cryptanalysis of Paillier encryption usually revolves around finding the private key by using the mathematical structure inherent in Paillier Cryptosystem, including the homomorphic formulas/properties. Since Paillier uses modular exponentiation (mod n^2), that could be represented as a lattice problem and if someone manages to solve the lattice, then it's a problem. However, since there are no known attacks that can do this, it's not currently a problem. There are key recovery attacks, where you try to recover the private key from known plaintext/ciphertext pairs, however this is kind of impossible now since there are no known ones, aside from the ones being printed when I run this in the console. They are encrypted, but these are pairs that someone will eventually unencrypt and then decode the rest of the messages. As for Paillier, the keys are hardcoded similar to the implementation of SDES and Triple SDES. This wouldn't be a problem as the numbers are very high, but

eventually someone would have enough patience to find the private key. Paillier is very well done, and I couldn't really find many weaknesses with it.

The first weakness in the implementation for QuickMAC is that there is no secret key for it that I could find, even retracing the code back toward MAC. Without a key in here, QuickMAC would be unable to provide additional authentication or authenticity aside from the session key that is sent earlier in the transaction before choosing decryptions. As for the hash function, it is a simple integer hash with bitwise operations and multiplication, meaning that brute force attacks can find out by generating a bunch of messages and by comparing their MACs, you can identify certain patterns and find out collisions between two certain plaintexts which create the same hash value, which you can use to give fake authenticity to a message. The action of creating new messages to find out the hash value is like the birthday paradox or what I learned is the birdbox paradox, as there are 13 birds and 12 bird boxes, meaning there will be at least one bird box with two birds. By applying this paradox to attacking QuickMAC, you will eventually find the hash value that belongs to two messages, breaching security. A faster way to do this method would be the chosen plaintext attack, where you send our certain plaintexts and observe the MAC that come out, then modify the next round of plaintexts to find out next version of MACs. If you can find similar hash values from how MACs interact with plaintext and cross reference with the plaintexts associated with their respective MACs, you'll be able to find the hash function's equations.

HMAC has a few weaknesses in the implementation, one being that it uses Sha-1. Sha-1 is weak and vulnerable to collision attacks, meaning if they find the hash from the sha-1 by finding two messages with the same hash values, then they would be able to forge their own HMAC. Attackers would send a valid message through the system and then send a malicious message

that would be accepted by using the same HMAC that was inferred from the valid message earlier. There are also hardcoded constraints in the code itself, being “36” and “5c”. If attackers can reverse engineer the code or gain access to source code itself like I have, then it would be easy to exploit the system. This is known as the known-key attack, which utilized the hardcoded constraints to inflict damage like gaining a HMAC key or creating new passwords with the HMAC hash and replacing them in the database with something that you can use. However, I have been unable to pull this off.

Aside from SDES and Triple SDES, this implementation is very secure, and I had a hard time trying to penetrate. It didn’t help that I didn’t know how to execute all these attacks and the ones that I have attempted have fallen short. The client and server both executed a SSL handshake that is executed using a Session key and a cryptographic algorithm utilizing the cipher suite. I am very thoroughly defeated by this implementation if it uses Paillier and HMAC since it utilizes multiple cryptographic algorithms. Thank you Niel Dass and Benjamin Avrahami for your implementation.

Raphael Chung