Implementing and Modifying Desai's UFE

Matt Fox

Armand McQueen

Andre Mroz

Eugene Oh

6.857

May 11, 2015

Abstract

One of the most important questions in security and cryptography today is how to determine if an encryption algorithm is strong enough to deter adversaries from learning the contents of private messages. Although there are many different protocols for testing the strength of an algorithm, one of the most stringent is IND-CCA (indistinguishability based on chosen cipher attack).

In lecture, we learned about UFE (Unbalanced Feistel Encryption), a protocol designed by Anand Desai that is IND-CCA secure. This protocol requires creation of a random bit string, which is passed through a CTR (Counter Mode) block cipher, xor'ed with the message, and then sent along with a CBC-MAC (Cipher Block Chaining - Message Authentication Code) that is xor'ed with the original random bit string. The total sent message is thus the size of the message plus the size of the random bit string. Our project proposal focuses on implementation, augmentation, and evaluation of this UFE protocol.

IND-CCA2 Security

Indistinguishability under adaptive chosen ciphertext attack (IND-CCA2) is a property of an encryption scheme. We can describe IND-CCA2 security as a game between and examiner and an adversary. The encryption scheme is IND-CCA2 secure if the adversary has a negligibly higher than 50% chance of winning the game. There are two phases to the game.

Stage I - "Find"

- The adversary may encrypt and decrypt anything they like
- The adversary creates two messages of the same length and sends it to the examiner.

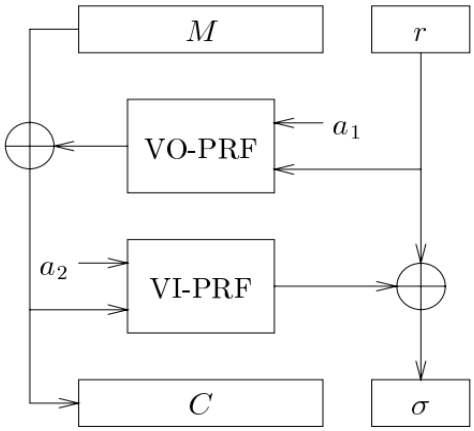
Stage II - "Guess"

- The examiner randomly chooses one of the two messages and encrypts it to create the challenge message. The challenge message is given to the adversary.

- The adversary may encrypt and decrypt anything except for the challenge message
- The adversary computes for polynomial (in the length of the message) time
- The adversary guesses which of the two messages was encrypted to make the challenge message

If the adversary guesses correctly, he wins the game

UFE Refresher



Desai's Unbalanced Feistel Encryption (UFE) is an example of an encryption scheme that is IND-CCA2 secure. At the heart of the scheme is the unbalanced Feistel structure, shown to the right.

The input to the scheme is the message *M*. A bitstring, *r*, is randomly generated for each message. The scheme is dependent on two pseudorandom functions (PRFs):

- The variable-output pseudorandom function (VO-PRF)
    - This outputs a pseudorandom bitstring that can be any size. The CTR mode of operation with AES as the block cipher is an example of a VO-PRF
- The variable-input pseudorandom function (VI-PRF)
    - This function takes in any size input and outputs a pseudorandom bitstring that is a fixed size. CBC-MAC is an example of a VI-PRF

During encryption, UFE:

- Randomly creates bitstring *r* as the source of randomness.
- Uses the VO-PRF to create the ciphertext, *C*, from *r* and *M*
- Uses the VI-PRF to create an encrypted *r*, sigma, from the ciphertext
- Sends both the ciphertext and sigma to the recipient

During decryption, UFE:

- Uses the VI-PRF to find *r* from sigma and the ciphertext
- Uses the VO-PRF to find *M* from the ciphertext and *r*

While this encryption scheme is IND-CCA2 secure, it requires transmitting both the ciphertext and sigma. In the standard implementation of UFE, the CTR mode of operation is the VO-PRF and CBC-MAC is the VI-PRF. This means that *r* and therefore sigma must be the size of the CBC-MAC which, with AES, is 16 bytes. For very short messages sigma can be much longer than the ciphertext, making UFE inefficient.

Project Goals

In our project we had a number of goals:

Implement Desai's UFE. We wanted to implement UFE using the CTR mode of operation as our VO-PRF and CBC-MAC as our VI-PRF.

Increase Data Throughput. We wanted to experiment with different modifications of UFE in order to send fewer bits while still maintaining IND-CCA2 security. If we were able to find a suitable modification, we would implement it.

Experiment with different VO-PRF. We wanted to examine and implement some other modes of operations and analyze whether they maintain IND-CCA2 security.

Analyze Performance. We wanted to analyze the performance of our different implementations to see what effect our changes had on data throughput.

Increasing data throughput by adding state to UFE.

We hoped to increase data throughput by decreasing the size of the sigma we needed to send across. Our first approach was to try to use a shared state between the two parties communicating. We then hoped to use this shared state to generate an *r* that both parties shared while only sending the ciphertext. However maintaining this shared state requires that

we assume messages will be delivered in-order and with no lost messages. We decided that this assumption was too strong and that we could make no assumptions about when and in what order messages would be delivered. Due to this, we must send the state along with the message and the state should be encoded – essentially sigma. Here using state gave us no improvement over the standard UFE and we abandoned this approach.

In paper [2], the authors found a way to transmit the state linked to a message while only sending the ciphertext and a small number of additional bits called the message number. The approach they used was to have a window, which is a small amount that the two parties can "fall out of synchronization". The decryptor receives the message and the message number and then uses his current state and the message number to find the state associated with the message. We feel that the purpose of UFE is to be highly secure and robust so we prefer to make no assumptions about how the states of the two parties may differ.

Revised Idea: Reduce Randomness

After deciding that a replacing the randomized value, R, with a state variable was infeasible given our design constraints, we decided to evaluate how the security of UFE compared to IND-CCA2 security. As stated in our first section, IND-CCA2 security is dependent on requiring an adversary use a number of encryption or decryption attempts that is greater than polynomial in the length of the message before having any advantage in guessing the correct message. This is of key importance, because we know that the boundary of how many attempts this requires is at the very least weakly increasing with the length of the message.

UFE, as described in the introduction of this report, has no change in the amount of security used to encrypt messages of different length. It follows that there is a different level of security provided for variable length messages when comparing to IND-CCA2 security. To make this clear, in the described UFE protocol, R is the number of bits of a block size - commonly about 16 bytes or so. If you have a message that is a single bit long, you still use 16 bytes worth of randomization to encode it. In the same vein, if the message is 1000 bytes long, you will use 16 bytes of randomization for the encoding. Clearly, when it comes time to

ask if the adversary needed to use more than polynomial accesses in the size of the message, you are asking very different questions with these two message lengths.

Now, it is important that we distinguish between absolute security and IND-CCA2 relative security. UFE provides the same absolute security to all message lengths. This means that any message encoded using UFE will be as secure as any other message regardless of the length (at least relative to the randomization factor). In real world applications, this is likely to be a much more important measure of security. Short messages and long messages alike need to be secure from feasible computational attacks, not only theoretical bounds.

However, we propose that creating a protocol that maintains IND-CCA2 relative security is an interesting and compelling problem. IND-CCA2 relative security means that if our protocol can be shown to be IND-CCA2 secure at a single message length, that it can be shown to be IND-CCA2 secure at all message lengths, and vice versa.

Now, let's look at how UFE differs from the IND-CCA2 relative security measure that we are using as our measuring stick. As shown in Desai's UFE paper, UFE is indeed IND-CCA2 secure up to some message size determined by the size of R used. Now, we have already shown that the number of accesses required to block an adversary in the security game of IND-CCA2 decreases with the message length. This means that UFE over-encrypts short messages by providing the same security to short and long messages.

In UFE, the level of security is equivalent to the number of bits of randomization. An attack on UFE would be to continue to encrypt the two test messages until you found a collision between an encryption and the provided ciphertext. This would happen if the same R value was chosen in an attacker's encryption attempt and the provided ciphertext's encryption. While this wouldn't allow the attacker to guess with 100% success rate (there theoretically could be a collision between the two different messages), it will give a higher than 50% success rate, beating the game. Now, the probability that a user gets the same R value as the test case is directly related to the number of unique R values. If R is b bits long, the chance of any single encryption using the same R value is 2^{-b}. Therefore, to reduce the expected number of accesses for the attacker, one would simply reduce the domain of R.

How large should the domain of R be?

This brings us to our central question - for a given message length, how large should the domain of R be? We spent a very long time discussing different ways of determining this boundary that would require just more than polynomial accesses given the length of the message. In the end, we discussed the problem with teaching assistants for the class and came to the conclusion that this was a P = NP type of problem.

The reason that this problem boils down to P = NP is that the key to IND-CCA2 security is finding where the number of accesses switches from polynomial time to non-polynomial time. This means that you have to find the number of accesses that is the last to be considered polynomial time and the first to be considered non polynomial. Unfortunately, this problem has yet to be solved in the realm of computer science, so we had to take another path.

As described before, we know that the function between message length and required size of R's domain must be, at the very least, weakly increasing. This is not formally proven here because it is not central to any arguments for our encryption algorithm. However, this can be seen easily by imagining a counterexample, where a smaller domain size is needed to protect a larger message. If we could encrypt a larger message with fewer options for R, then we could simple encrypt all smaller messages and use 0's or some other constant factor to encrypt it as a larger message. This would break the fact that you need a larger domain for a smaller message length, showing that the function must be weakly increasing.

Now, it is fairly simple to see that we can use our shortening of the domain of R to our advantage. A very simple way to reduce the domain of R would be to create a randomization factor with fewer bits. The advantage to using a randomization factor with fewer bits is that the associated information that is sent between the sender and the receiver of a message is shorter as well. In UFE this value is called sigma and is the same number of bits as R.

The fact that we are decreasing the size of sigma is also integral to our design because this is the main motivating factor to use our modified version of UFE instead of Desai's version. Sending fewer bits across for our sigma value will increase the message to total-data-sent

ratio, making this implementation more efficient. As you will see in the results section, this also increases total throughput in our implementation.

Modified UFE

Although we have now made the argument for reducing the number of bits used in UFE's random variable, we must explain how this will factor into the VO-PRF's functionality as well as the creation of the sigma variable. This was the main area for design in our project, and we went through many options before we found and decided upon a fairly simple end result.

Firstly, we realized that it is extremely important to keep the Feistel ladder form of UFE in order to keep this as a modification of UFE and to maintain the possibility of IND-CCA2 security. This can be seen because we must have some way of securely encoding the bits of randomization that can be both encoded and decoded, and the CBC-MAC worked perfectly well as a VI-PRF in this function.

Next, we knew that we needed to keep the input to our VO-PRF and the output from our VI-PRF as the same number of bits as the original R from UFE. This idea developed mainly because the available libraries for the common modes of operations and MAC protocols take very standard block sizes, but also because encrypting using a VO-PRF with a variable block size significantly complicates the encoding step, which would reduce throughput and negate the efficiency gains that our proposed system generates.

Therefore, we had to develop a mapping between the smaller bit size our new R, from here on called R_modified, and the original R, from here on called R_original. Our first idea in this space was to create and share a lookup table for the sender and the receiver. R_modified could then be used to index into this table and define the R_original which was used for encryption and which should be used for decryption. However, this would take a large amount of memory on both sides of the transaction and would require a ton of overhead to set up any secure connection.

Our next and final idea was to simply pad R_modified with enough zeros to get to the length of R_original. This would allow us to have no shared memory or lookup table to communicate while still maintaining a smaller bit footprint for sigma. To create sigma, we simply calculate the CBC-MAC of the ciphertext and then cut off the lowest order bits until it had the same number of bits as R_modified. This design can be seen in Figure 1 below.

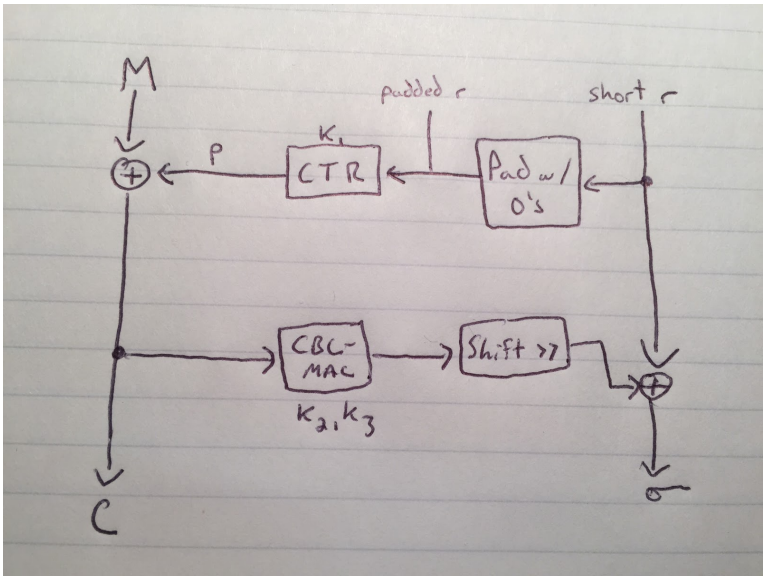


Figure 1: A diagram of our modified UFE design.

Now, these design decisions merit some discussion because, at first glance, they seem to introduce some regularity into the system that could reduce security. The first instance where this seems to occur is in padding R_modified with zeros at the end in all cases. One could imagine that running values that all end in the same number of zeros will produce related results. However, a key feature of a variable output pseudo-random function is that the input to output mapping is not easily related. This means that we lose no security by mapping numbers that end in zeros compared to the same amount of randomly generated numbers.

Furthermore, CBC-MAC is a variable input pseudo-random function. This means that the top bits of the CBC-MAC are just as random as the bottom bits. Moreover, there is no way to change the ciphertext in such a way that you change only the bottom bits of the CBC-MAC in a predictable way. The pseudo-randomness for both CTR and CBC-MAC come from the use of an ideal block cipher in the encryption step, which will be discussed more later.

We have skirted around the fact that our implementation must define and use some sort of function between message length and bits of randomization used. As shown above, we cannot define this function explicitly because it is, at its core, a P vs. NP problem. However, given that we want to implement this protocol, we have to choose some function that could hold some properties that are similar to the real boundary.

Our protocol allows the user to define the function between message length and R_modified length. This will allow any application to prioritize security or higher data efficiency. In our implementation described below, we have decided to define a constant ratio between message length and R_modified length. This function is monotonically increasing which fits the bill as at least weakly increasing. Furthermore, it was quite simple to implement and did not significantly increase computation time.

Our version of the modified UFE protocol allows variable sized messages to be sent in the same connection. This means that we must have a method for communicating how many bits of randomization the decryption must use to correctly compute the decryption. Luckily, a property of using a weakly increasing function is that you can uniquely identify message length and R_modified length from the ciphertext just by analyzing total length that was computed by the user-defined function. In our implementation, we know that the total ciphertext length (ciphertext plus sigma) will be equal to 1+ R/M times the original message length, where R/M is the ratio between R_modified length and message length defined at the outset of the protocol.

Now that our protocol is fully defined, we have also implemented an example in order to compute performance testing against the original UFE. The next section is devoted towards describing the details of this implementation.

UFE Implementation

As to the actual code implementation of UFE, we imported an existing Python library named “pyaes” that has an implementation of the block cipher AES as well as several of the common

modes of operation. Using this library, we implemented our modified UFE as a class that was initialized with several inputs as shown below in figure 2.

```
class UFE:
    def __init__(self, modeOfOperation, key1, key2, key3, modifiedUFE=False, m2rRatio=16):
        self.modeOfOperation = modeOfOperation
        self.k1 = key1
        self.k2 = key2
        self.k3 = key3
        self.modifiedUFE = modifiedUFE
        self.m2rRatio = m2rRatio
        self.blockSize = 128
```

Figure 2: A code snippet of how the UFE class is initialized.

The input modeOfOperation is a string that specifies what mode of operation to use as the VO-PRF in our modified UFE. The three keys are the keys of the user that will be used in the encryption blocks of the VO-PRF and the VI-PRF, which in our case are a block cipher mode of operation and the CBC-MAC. The boolean modfiedUFE specifies whether or not the class should be initialized to use our modified version of UFE or to use the version of UFE that was described in lecture. The final input, m2rRatio, is the bit ratio of the length of the message to the length of R, which is the factor of randomization. Lastly, we set the default block size 128 bits (16 bytes).

Contained within the class are the functions ***encrypt*** and ***decrypt***, which do the actual encryption and decryption of a message using the settings the UFE is initialized with. The encryption returns the ciphertext and sigma as a tuple of the form (ciphertext, sigma). Meanwhile, the decryption takes the ciphertext and sigma as inputs. Additionally, the class also contains several helper functions that are used by the ***encrypt*** and ***decrypt*** functions in order to carry out the encryption and decryption. A couple of the more relevant ones include the function ***cbc_mac*** (shown below)***,*** which calculates the CBC-MAC of the ciphertext that is used in the UFE encryption scheme.

```
# Takes in message as unicode string, outputs CBC_MAC as a list of bytes (in integer form)
def cbc_mac(self, ciphertext):
    aes1 = pyaes.AESModeOfOperationCBC(self.k2)
    # convert message to bytes
    #blocks = [ ord(c) for c in ciphertext ]
    blocks = self.split_ciphertext_into_blocks(self.string_to_bits(ciphertext))
    #ciphertext = aes.encrypt(plaintext_bytes)
    n = len(blocks)
    for i in range(n-1):
        nxt=aes1.encrypt(blocks[i])
    aes2 = pyaes.AESModeOfOperationCBC(self.k3,iv = nxt)
    nxt=self.bits_to_bytes(self.string_to_bits(aes2.encrypt(blocks[n-1])))
    return nxt
```

Figure 3: Implementation of the generation of the CBC-MAC

In the above function, we first find the CBC-MAC of the ciphertext using the second key that is provided with the initialization of the class. Then, we encrypt the output of the CBC-MAC using our block cipher with the third key provided by the user.

Another important function in the UFE class is ***generate_r*** which generates an R for the UFE based on the m2rRatio given when the class is initialized. The function returns the shortened R and also a padded R to match the block size that is then passed to the encryption blocks. The function ***generate_r*** is shown below in figure 4.

```
# returns a list, first element is r without padding represented as list of bits
# second element is r with padding represented as list of bits
def generate_r(self, message):
    result = []
    messageBitArray = self.string_to_bits(message)
    if self.modifiedUFE:
        lengthOfR = int(math.ceil(len(messageBitArray)/self.m2rRatio)) #get length of R
        if lengthOfR > 128:            #R can't be longer than block size
            lengthOfR = 128
    else:
        lengthOfR = 128
    rand = random.getrandbits(lengthOfR)   #generate a random bitstring of lengthOfR bits
    rand = self.int_to_bitlist(rand)       #convert rand to a list of bits
    for item in rand:
        result.append(item)
    while len(result) < self.blockSize:
        result.append(0)               #pad R while it's length is shorter than the block size
    return result,rand
```

Figure 4: Implementation of the function that generates R and padded R

Due to some of the functions that we imported, such as the generation of random bitstrings, we also had to implement several helper functions within the UFE class that converted between different representations of bits. These helper functions convert between a Python list of bits, integers, and a Python list of bytes.

UFE with Different Modes of Operation

Not only did we want to test the performance of our modified UFE, we also wanted to test the UFE described in lecture while trying different block cipher modes of operation. The three main modes of operation we tested were CTR, CBC, and CFB. We wanted to compare the performance of each of these modes as well as show that each still preserved the IND-CCA2

property of the UFE encryption scheme. The following arguments are all under the assumption that the AES block cipher we are using is pseudorandom.

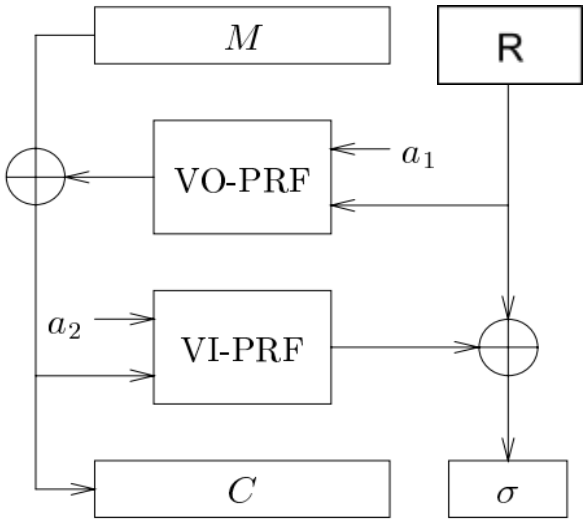


Figure 5: The UFE Scheme

We decided to keep the VI-PRF as a two-key variant of CBC-MAC for every UFE we implemented with the different modes of operation due to time constraints. In this variant of CBC-MAC, we compute the CBC-MAC of the ciphertext. Then we take the output and apply another block cipher to it using a second private key. Petrank and Rackof have already analyzed this variant of the CBC-MAC and shown it to be secure on variable length inputs.

Looking at the scheme of UFE in figure 5 above, we can see that the scheme will remain IND-CCA2 as long as R is a randomly generated initial bitstring. An adversary will not be able to generate an R' that is related to the original R by changing the ciphertext in any way due to the nature of the VO-PRF and VI-PRF. Additionally, the scheme is only invertible to retrieve R given a ciphertext and sigma if the secret key (a_2) is known.

Because of the security of the scheme, the only thing left to show in order to prove that UFE remains IND-CCA2 secure using any of CTR, CBC, and CFB is that those three modes are VO-PRFs. CTR gives variable length outputs because the number of blocks can vary depending on the size of the message and an implementation can add more blocks simply by

incrementing the counter. Given that the initial counter, in this case R, is a randomly generated bitstring, the output of CTR will be pseudorandom because of the pseudorandomness of the underlying block cipher, AES. The output of CBC and CFB can also vary in length based on message size because more blocks can simply be chained together as message size grows. CBC and CFB are also both initialized using a randomly generated IV and so the pseudorandomness of the underlying block cipher, AES, will also make the output of these two modes pseudorandom.

Performance Analysis

Our first objective with this project was to attempt to substitute in other modes of operation than CTR into the "VO-PRF" portion of Desai's UFE protocol and compare them. As discussed above, we successfully substituted CBC and CFB while maintaining security. To determine how quickly each mode could perform encryption and decryption, we created a trial test where we put a randomly generated 160-byte message through an instance of UFE with each of the three modes. We repeated this trial 5000 times. All of the modes performed similarly over many trials, though CBC and CFB were shown to be ever so slightly faster than CTR. However, when the length of the message increased, the amount of time the encryption phase of CBC and CFB took increased dramatically because they cannot be run in parallel, unlike that of CTR.

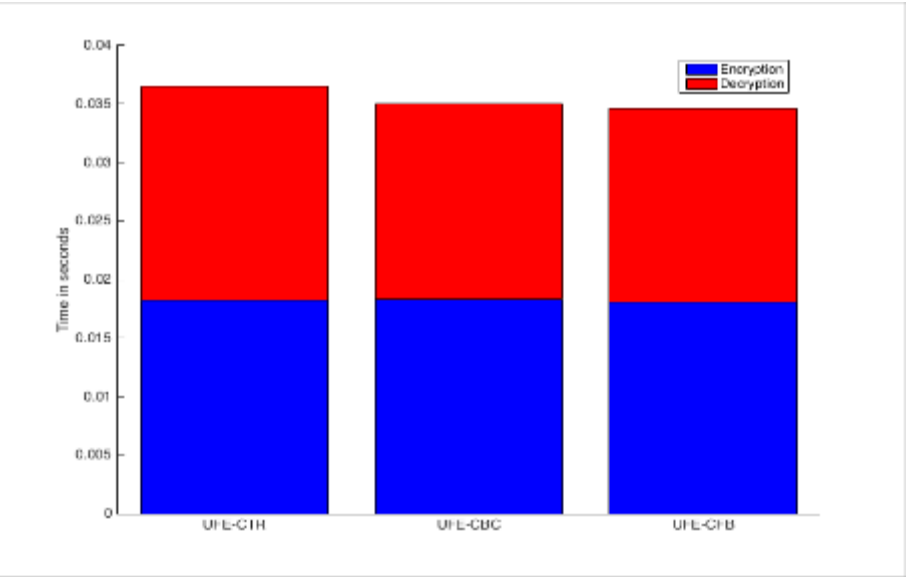


Figure 6: Encryption and Decryption times for the modes of operation.

After testing each of the modes, we needed to test the effect our modified UFE would have compared to UFE at different values of the message bits to r ratio. We predicted that due to needing fewer bits of unpadded r, for a larger ratio there would be fewer xor operations and the total amount of time would decrease. In our implementation, the function that generates r is standardized between the different types of UFE, meaning that there should be a negligible time differential in this step due to having almost the same number of operations. Our test, which was similar to the test we used for the different modes of operation except that only 500 trials were used, was able to confirm our hypothesis, and larger values of the ratio could reduce time in both the encryption and decryption steps by a fairly noticeable amount.

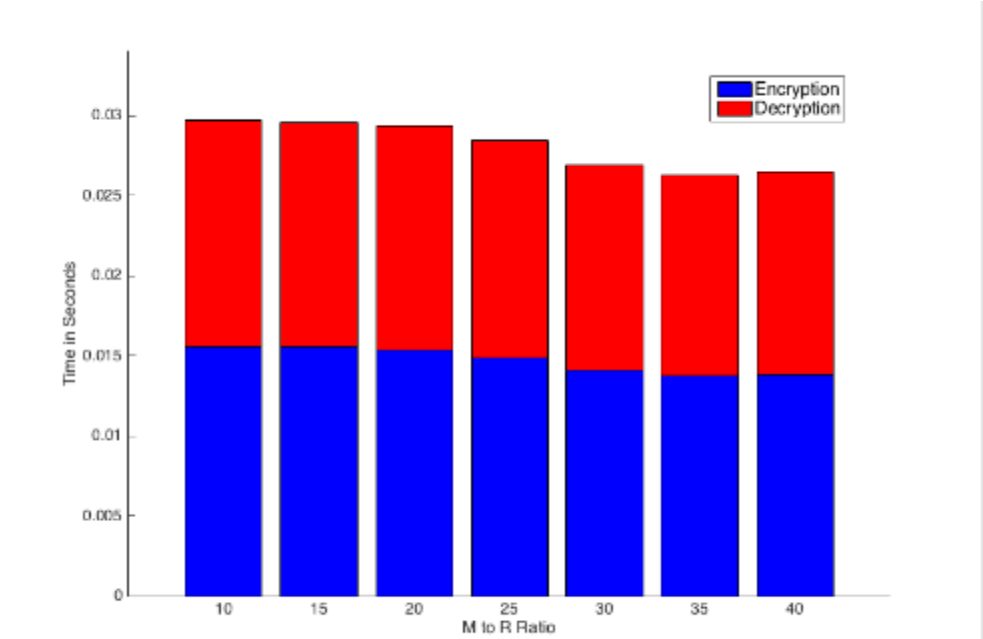


Figure 7: The effect of changing our message bits to r ratio.

In addition to shortening the amount of time it takes to encrypt and decrypt a message, when used in a protocol that transfers encrypted data between two entities, fewer data bits are sent in all, and faster, which causes the throughput of the protocol to increase. This improvement can be scaled with the size of the message itself by varying the ratio. The user sets a personal level of security in relation to the message length that should still ensure that it is secure under IND-CCA2. While we are varying the absolute defined level of security (being the ratio between the length of the message and the length of r), we maintain that it would take more than a polynomial number of encryptions to break the security. This means

that, throughput-wise, our modified UFE can never perform worse than regular UFE. This idea can be shown in Figure 8 below.

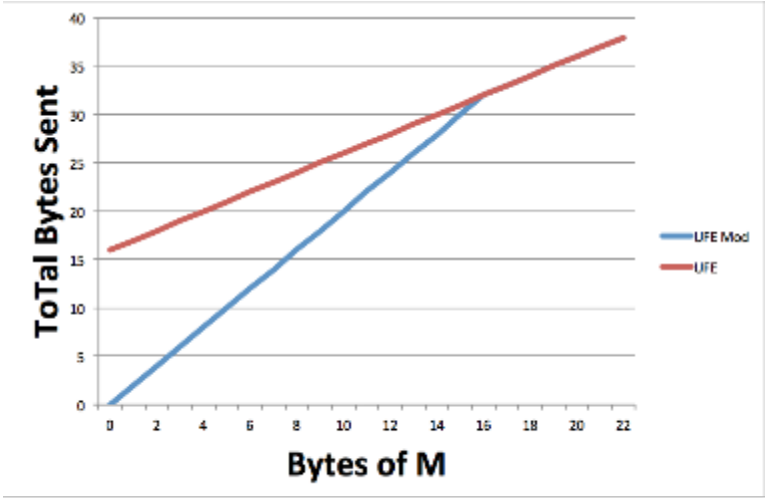


Figure 8: Comparing the total number of bytes sent in each UFE.

Conclusion

Desai's UFE is a widely accepted IND-CCA2 encryption algorithm that has been used frequently in both academic and commercial implementations. We believe that the modified forms of UFE that we have discussed throughout this paper are both interesting and informative in the light of IND-CCA2 security. Although the modified form of UFE which we implemented can theoretically maintain IND-CCA2 security, we would not recommend a commercial system use this as security for short messages as the real world has a bound of reasonable computational power, not a strict limit on adversary accesses. Despite this drawback of our algorithm, the modified UFE provides an interesting look into maintaining a tight bound on IND-CCA2 security while maximizing data efficiency.

Bibliography

[1] Desai, Anand. "New Paradigms for Constructing Symmetric Encryption Schemes Secure against Chosen-Ciphertext Attack." Advances in Cryptology — CRYPTO 2000 Lecture Notes in Computer Science (2000): 394-412. Web.

[2] Trostel, Jonathan. "CMCC: Misuse Resistant Authenticated Encryption with Minimal Ciphertext Expansion." (n.d.): n. pag. Web.

[3] E. Petrank and C. Rackoff, “CBC MAC for Real-Time Data Sources,” Dimacs Technical Report, 97-26, 1997.