A Very Lightweight Encryption Method Based on Multiple Substitution Tables

L Henschen* and J Lee

Electrical and Computer Engineering, Northwestern University, Evanston, Illinois, 60208, USA

*Email: henschen@eecs.northwestern.edu

Abstract. We present a very lightweight encryption method based on multiple substitution tables. The method does not use sophisticated integer arithmetic and is therefore suitable for small embedded systems and other applications using simple processing elements. Plain text is encrypted one byte at a time. Each byte is replaced by the substitution element in the current table, and the plain text byte also determines the table to be used for the next byte. Encrypting and decrypting one byte involves only a table look-up, no arithmetic or bit manipulation. The same plain text byte has very small probability of being encrypted the same way twice in a row. We also present additional techniques for making the method secure against attacks when used in applications where the messages may have special or restricted forms.

1. Introduction

There is an increasing need for lightweight encryption systems that are private and have a high-level security. [1] shows the importance attached to lightweight cypher development by NIST. [2-6] offer interesting surveys of techniques and the challenges for lightweight encryption in embedded systems (ES) and the Internet of Things (IoT).

Many embedded systems have components that communicate wirelessly with each other but with only one node that communicates with the outside world. Examples include patient monitoring, home control systems, building security systems, and many others. The internal messages must still be secure from attack from the outside world. For example, the patient monitoring system would likely communicate information to a hospital or doctor’s office through a relatively powerful processing device such as a PC/laptop or cell phone, but the nodes on the patient’s body only communicate with each other. Such systems, then, need two separate kinds of encryption – general encryption for communication with the internet and a private encryption system to protect its internal messages. The latter must be lightweight because the processing elements in many of the embedded systems components are not powerful enough for the more common public encryption methods.

The method described here has many interesting properties. For example, a plain text byte will not always be encoded the same way in the cypher text. A message that is transmitted twice has a very small probability of being encrypted the same way both times. The method is private. The tables and the various coordination mechanisms are meant to be loaded into the components of the system directly rather than passed through some public medium the way public keys are often distributed. There is no public information that would allow an attacker to encrypt a message, and therefore many of the common methods of attack, such as chosen plaintext attacks [7] in which the attacker encrypts carefully selected plain text messages, are not possible.
Public key encryption systems, such as RSA [8], are the most widely used methods for general internet encryption. They are based on integer arithmetic with large numbers and large primes. These operations require a high level of computing power and take non-trivial amounts of time for encryption and decryption. Most ES have real-time requirements, and the processing elements used do not have enough computing power to perform the calculations in a reasonable amount of time. Stream and block ciphers, such as those described in [2-6] and many others, encrypt one bit or byte or block at a time by typically XORing each plain text unit with a mask, or key. The key is derived from an initial key by a combination of bit-shifting and masking operations. Encryption and decryption still require non-trivial time. Such systems are also susceptible to many common key-discovery attacks. The method described here does not use keys and is not susceptible to key-discovery attacks. It has no public information, as in RSA. It uses no complex arithmetic or bit manipulation and can be implemented on the tiniest processing elements.

Section 2 describes the method in detail. Section 3 contains several ideas for blocking attacks in situations where messages might have special properties or formats. Section 4 contains our concluding remarks.

2. The Multi-table Method MTM
Section 2.1 describes the data structures and algorithms for encryption and decryption. Section 2.2 discusses the generation of the tables. Section 2.3 discusses security and makes other comments.

2.1. The Multi-table Method
The sender and receiver each have the same array T of tables. Each table is an array with three columns (labelled encrypt, decrypt, and next) and 256 rows. The rows are accessed using a byte as subscript – a byte of the plain-text during encryption or a byte of the cipher text during decryption. Let b be a byte and t be a table in T. The values in the columns for row t[b] are:

- encrypt – the replacement byte b’ for b for this table.
- decrypt – the decryption byte for b in this table (that is, the byte b” for which t[b’].encrypt = b).
- next – an index to another table in the array.

Each byte occurs exactly once in the encrypt column for a table t. Therefore, the encrypt column represents a one-to-one mapping of bytes to bytes. The decrypt column represents the inverse of this mapping. That is, if t[b].encrypt = c, then t[c].decrypt = b. Thus, for every byte b, t[t[b].encrypt].decrypt = b. Figure 1 illustrates the format. Space precludes showing complete 256-entry tables. The values used in these partial tables are chosen so that Example 1 may illustrate several interesting features of the method. The entries shown here are formatted as encrypt, decrypt, next. For example, in Table 0 the encrypt byte for index 0 is 3, the decrypt byte for that entry is 6, and the next index is 2.

| Index | Table 0 | Table 1 | Table 2 | Table 3 |
|-------|---------|---------|---------|---------|
| 0     | 3, 6, 2 | 4, 21, 0| 8, 1, 2 | 14, 3, 1|
| 1     | 2, 9, 0 | 1, 1, 3 | 0, 0, 1 | 7, 64, 1|
| 2     | 4, 1, 7 | 7, 3, 21| 3, 3, 0 | 5, 12, 1|
| 3     | 7, 0, 3 | 2, 37, 2| 2, 2, 3 | 0, 37, 3|

Figure 1. Portion of an array of tables.

The table entries for the encrypt and next columns should be chosen randomly so that no simple patterns exist. See Section 2.2 for more discussion on table generation. Note that the tables are generated and distributed to the sender and receiver before message passing begins. Therefore,
sophisticated means can be used to generate the tables to ensure that the tables and resulting encryptions appear random, and there is no decrease in performance at message transmission time.

The sender and receiver also each have the same array, $ST$, of starting table numbers. When the sender encrypts a plain text message, it selects the next starting table number as the first table to use in the encryption process, as described in the next paragraph. When the receiver decrypts the cypher text, it also selects that starting table number. The first message sent uses $ST[0]$ for the starting table; successive messages use $ST[1]$, $ST[2]$, etc.. Each new message uses the next element of $ST$, with wrapping after the last element has been used.

The cypher text is formed as follows. First, the sender selects the next element, $n$, from $ST$ as the starting table number. The corresponding cypher text byte is the replacement byte for $b$ in table $T[n]$. Then, before proceeding to the next plain text byte $n$ is replaced by the next table index corresponding to $b$ in table $T[n]$. The full algorithm is shown in Figure 2.

To decrypt the cypher text, the receiver selects the same element, $n$, from $ST$ as the starting table number. Let $c$ be the first byte in the cypher text. The corresponding byte of the decrypted text is the decryption byte, $d$, for $c$ in table $T[n]$. Note that during encryption the next table index is in the entry for the original plain text byte, i.e., $d$. Thus, $n$ is replaced by the next table index in $T[n]$ corresponding to $d$ (not $c$). The full algorithm is shown in Figure 2.

```c
encode(msg) {
    int n = ST[encSTindx++];
    int j = 0, L = length(msg);
    while(j<L) {
        c = msg[j];
        transmit (T[n][c].encrypt);
        n = T[n][c].next;
        j = j+1;
    }
}

decode(msg) {
    int n = ST[decSTindx++];
    int j = 0, L = length(msg);
    while(j<L) {
        c = msg[j];
        decode[j] = T[n][c].decrypt;
        n = T[n][decode[j]].next;
        j = j+1;
    }
}
```

**Figure 2.** The encryption and decryption algorithms.

**Example 1.** We illustrate the encryption and decryption processes on a few sample byte strings using the tables from Figure 1. Figure 1 shows only a small portion of four tables, so the sample messages in this example use only a few values.

Sample message 1: Suppose the plain text message is “113”. If $T[2]$ is chosen as the starting table, then the encrypted string is “010”. The entry at $T[2][1]$ (1 is the first byte of the plain text message) is (0, 0, 1). The encoding for this occurrence of 1 is 0, and the table to use for the second byte of the plain text message is $T[1]$. The next plain text message byte is also 1, but this occurrence of 1 gets encoded as 1 because the entry at $T[1][1]$ is (1, 1, 3). The last byte of the message is 3, and $T[3]$ is used to encode it. The entry at $T[3][3]$ is (0, 37, 3), so the encoding is 0.

Decoding the above cypher text: The decoder also starts at table 2. The cypher text is “010”. The entry at $T[2][0]$ is (8, 1, 2). The decrypt component is 1, so the first byte of the decoded text is 1. During encryption the next table pointer was the one at position 1 in table 2; that is, during encryption it is the original plain text byte that is used to reassign the table number. In this case, the next table pointer at $T[2][1]$ is 1, so the next table to be used for decryption is $T[1]$. The second cypher byte is 1. The decrypt byte at $T[1][1]$ is 1, so the next decoded byte is 1. Again, it is this decoded byte that is used to find the next table index. In this case the next table index is 3. The
last cypher byte is 0. The entry at T[3][0] is (14, 3, 1). The decrypt component is 3. So, the decrypted text is “113”, i.e., the original plain text string.

Sample message 1 with a different starting table: Suppose the starting table was chosen to be T[0]. The encoding would be “227”. In this case, T[0] is the only table used. This is an illustration of a poorly chosen set of values for encryption and next table.

Sample message 2: Suppose the message was “312”, and the starting table was T[0]. The encoded message is “777”. This is a curious example because it shows that a message with all distinct characters could produce an encoding in which all the encoded bytes are the same.

**Theorem 1.** Assume encode and decode use the same array T and starting table number n. Let M be an arbitrary string of bytes. Let C be the string produced by encode(M), and let M’ be the string produced by decode(C). Then M = M’.

**Proof:** The proof is by induction on the length, k, of M.

Base case: k = 1. Let the single byte in M be m. C consists of the single byte c = T[n][m].encrypt. By hypothesis, decode will also use table T[n] to decrypt c. By the construction of the tables, the decrypt component of T[n][c] is m.

Induction case: Assume the theorem holds for strings of length k, k >= 1. Let M = mM*. Let cC* = encode(mM*). When decode is applied to cC*, table n is used to decode c. By construction of the tables, the result is m. After encrypting m, encode sets the next table number to T[n][m].next. The decode function also sets the next table number to T[n][m].next. M* is a string of length k, and encode and decode both use the same table to begin encryption of M* and decryption of C*. By the induction hypothesis, decode will produce M*. Thus, the decoded string is mM*.

**QED**

2.2. Table Generation

The tables are generated once and before any message transmission takes place. This means that as much effort as desired can be aimed at creating tables with good properties, and any such work done in creating the tables will have no effect on the work required for encryption and decryption. Computationally demanding pure random number generators or sophisticated methods for random permutations, such as Knuth Shuffle [9], can be used for generating the encrypt values. Pure random number generators can be used to generate values in the range 0-(N-1) for next table pointers. The generation process can continue where it left off after each table is generated, or a new seed can be used for each successive table. As already noted, considerable time can be spent in table generation with no effect on performance at message encryption/decryption time.

Table generation may also include an evaluation and modification phase in which the table set is tested for performance. Sample strings can be encrypted. Tests can be performed to check for sequences of cypher text that have bad properties, excess use of some of the tables over the others, circularity of paths through the tables, etc. This could be particularly useful for applications in which the messages have special properties not present in ordinary text; examples include formatted sensor node reports in a sensor network, message traffic in a smart home, XML-formatted messages, and others. Again, such testing does not affect the performance during encryption and decryption because it is done before the tables are distributed and message passing begins.

We conjecture that randomness in table generation provides a kind of non-linearity like that obtained by the use of non-linear combining functions [9] in various stream methods.
2.3. Comments and Notes
First, as already noted, the multi-table method is extremely fast and simple to implement. Other stream methods spend time during the encryption and decryption processes to generate successive keys, typically 10 or more rounds with each round involving bit manipulation and table lookup (see, for example, [10] or [11]) The multi-table method simply requires an array look-up.

Like other stream ciphers, a byte b has only a small probability of being encrypted the same two times in a row. If we assume that the tables are generated so that no byte is encrypted the same way in two different tables, then the only way two successive occurrences of b can be encrypted the same way is for the sequence of tables used to encrypt all the bytes between the two occurrences to lead back to the same table used in the first encryption. That is, if the plain text has a sequence bc1c2…cnb and if table n is used to encrypt the first occurrence of b, then after encrypting cn the next table would have to be n again. If the next pointers are generated truly randomly, the sequence of tables used during encryption is essentially a Markhov chain. The table to be used next after encrypting a plain text byte b does not depend at all on how many bytes precede b in the plain text or what sequence of tables was used to encrypt them. The probability that the next table would be n should be approximately 1/N, N the number of tables in the array. A corollary is that successive occurrences of letter sequences, like “the” or “all”, also have probability only 1/N of being encrypted the same way. This probability can be significantly lowered by using techniques described in Section 3 that obfuscate the original plain text or distribute frequently occurring letters among non-printable bytes.

Unlike other stream ciphers, a message M that is transmitted multiple times has only a 1/N probability of being encrypted the same way two times in a row. The encryption depends only on the selection of the starting table from ST, and elements of ST are random in the range 0-(N-1). If two messages have identical initial segments, two successive encryptions have only a 1/N probability of that initial segment being encrypted the same way. (The presence of identical initial segments was used to break the ENIGMA code in WW II [12].) The probability of duplicate encryption can be significantly lowered by techniques discussed in Section 3, such as random initial padding. Many encryption systems, notably RSA, encrypt two occurrences of a message M the same way, and this feature can be used by attackers to try to break the code.

Most of the attack methods against other encryption systems are not relevant for MTM. For example, most attacks are aimed at discovering the key. Knowing the private key in an RSA-type system, for example, allows attackers to directly decrypt cyphertext. Knowing the key in a stream cypher system allows an attacker to explore different algorithms that senders and receivers might be using for generating the key sequence. There is no key in MTM in the normal sense used in cryptography. Even knowing the starting table number for a particular transmission doesn’t help an attacker because the next table pointers are random and depend on the corresponding bytes of the plain text. Another common attack strategy, the chosen-cyphertext attack [13], encrypts carefully chosen plain text messages and then analyses the results. This is not possible in MTM because the attacker cannot encrypt – the attacker doesn’t know what the tables are. Similarly, “semantic security” [14] is meaningless for private encryption methods. Application of semantic security requires at a minimum the ability of an attacker to know the encryption of some specific strings, which of course an attacker cannot know in a completely private encryption system.

MTM displays both confusion and diffusion, two important concepts introduced by Shannon [15]. Roughly speaking, confusion means that the cypher text depends in some complex non-linear way on the plain text. Confusion is a feature of all substitution methods. The examples in Section 2.1 demonstrate a high degree of confusion. In particular, the last example, in which the cypher text consisted of all the same byte, shows a high degree of confusion. Diffusion means that changing a single byte or even bit of the plain text produces large changes in the cypher text. Changing a single bit in some byte of plain text means that a different entry in the current table will be used to produce the cypher text byte. By construction of the tables, this cypher byte will be different than and bear no recognizable relationship to the byte that would have appeared if no change had been made. Even more, with probability (N-1)/N the next table will also be different, and therefore the next byte will also
almost surely be different. Successive cypher bytes will be different until a point at which the same table used in encrypting the original plain text is used to encrypt the modified plain text. Thus, changing even a single bit in the plain text will normally produce huge changes in the cypher text.

The multi-table method does have other features in common with stream methods. For example, MTM is susceptible to bit insertion and byte modification attacks. In these attacks the receiver will use an incorrect byte for table look-up, and the resulting decryption byte will be wrong. Even worse, the next table pointer will be wrong, which means the following bytes will also be decrypted incorrectly as described in the preceding paragraph. For messages that have specific formats and information content, the receiver might be able to recognize the point at which the corruption occurred and possibly recover. However, at present we do not have a general self-correcting technique, unlike the case for certain kinds of stream cyphers. While these kinds of attacks do not help an attacker decode messages, they can disrupt the business of the sender/receive pair. We note that the inclusion of integrity checks, such as CRC checks, can at least allow the receiver to determine that the message has been corrupted and to take some recovery action. On the other hand, like stream cyphers, MTM is very easy to implement and can therefore be implemented and deployed as a separate hardware component between the sender and its transmitter or between the receiver and its stream input. In this way, the encryption tables can be kept secret even from the sender and receiver.

3. Additional Techniques Specifically for ES and IoT Applications

Message sets in embedded system applications may have properties that make encryption less secure. For example, in a home control system the message set may be small – a limited number of commands (turn AC off, open bedroom vent, etc.) and a limited number of sensor readings (temperature in room x is y, window in room z is open, etc.). Attackers might be able to gain information about the encrypted messages based on knowledge about the limited message set. We present three techniques that can help block such attacks.

3.1. Spreading the Most Frequently Occurring Characters in Text Messages

In general text the blank occurs several times more often than even the most frequently occurring English language letter, ‘e’. Then certain letters (‘e’, ‘t’, ‘o’, etc.) occur much more frequently than others. Blank occurs so much more often than the others that the various encryptions of blank in the different tables in the table array also occur much more frequently than other bytes in the cypher text. For example, in one experiment a normal text message of 1343 bytes contained 194 occurrences of blank, 114 occurrences of ‘e’, and 87 occurrences of ‘o’. After the 10 most frequently occurring letters, most of the remaining characters occurred only a handful of times. In an experiment with 16 tables (less than recommended), the 16 different encryptions for blank occurred in the cypher text from 7 to 18 times; for example, one table encrypted blank as 205, and this encryption was used 18 different times. The byte 205 appeared in the cypher text two additional times (because two tables encrypted some non-blank character as 205) for a total of 20 different occurrences in the cypher text. The different encryptions of the letter ‘e’ occurred from 2 to 20 times. No other encryption occurred more than 11 times. Thus, an attacker could look at the most frequently occurring cypher bytes and guess with fair accuracy that these were blank or ‘e’. This information might give the attacker strong clues as to word boundaries and the occurrence of certain actual words (e.g., “the”, which has an ‘e’ followed by a blank). A similar situation occurs with XML messages. The alphabet for these messages is already limited – blank, angle brackets, perhaps only lower-case letters, digits, ‘=’, and ‘.’. Again, brackets and blank would likely occur way more often than all the other bytes. Note, using more tables reduces risks associated with letter occurrences. In an experiment with 256 tables, no encoding of blank occurred more than 4 times in the sample text mentioned above.

For plain ASCII text messages, we may make use of the bytes in the range 128-255 because all the ASCII characters are represented by bytes in the range 0-127. Spreading uses the bytes 128-255 as alternate representations of the most frequently occurring plain text message bytes. A simple implementation might take, say, the top 10 most frequently occurring characters in the message set and use 12 consecutive numbers from 128-255 as alternate representations. For example, blank might
be represented by 32 (the normal representation of blank) and any of the numbers 128-139. The plain text is pre-processed by leaving the first occurrence of blank alone, replacing the second occurrence by 128, the third occurrence by 129, etc., repeating the set 32 and 128-137 as necessary until all blanks had been processed. Similarly, the letter ‘e’ might be represented as 101 (the normal representation for ‘e’) and numbers in the range 140-151. Spreading the most frequently occurring characters means that in the modified plain text no character occurs many times more often than others and there are 30-40 characters that occur a similar number of times. This is also reflected in the cypher text, and an attacker would have much less chance of guessing where blanks or other frequently occurring characters are. In our experiment with spreading the top 10 characters in the example with 16 tables, no representation of blank occurred more than 15 times, less than encryptions for the letter ‘h’, which was the 11\textsuperscript{th} most frequently occurring character and not spread.

3.2. Padding
Random padding is a well-known strategy. It is not unique to MTM, but it can be especially helpful in embedded systems applications. Random initial padding means insertion of random bytes at the beginning of the plain text before encryption. In general, this strategy guards against an attacker using knowledge about the beginning of the plain text messages. Random initial padding reduces the probability that a given message m will be encrypted the same way in different transmissions to well below 1/N mentioned in Section 2.3. Random bytes can be added at the end of the message to disguise the length of the message. In many embedded applications the message set is small, and the sizes of the messages may be known to an attacker. Such information could be used to at least to distinguish one kind of message from another because the length of the cypher text is the same as the length of the plain text. Finally, random bytes could be inserted throughout the message, further obscuring the plain text message.

When the plain text bytes are distributed throughout the range 0-255, the sender and receiver have to share information about how initial, terminal, and distributed padding is done. This can be accomplished through additional tables, like T and ST, loaded when the system is first deployed. If byte values in the plain text message set are restricted, random padding bytes can be selected from the unused bytes. For example, padding bytes can be chosen from 128-255 for ASCII text messages because ASCII characters are all in the range 0-127. In such cases, there is no need for additional coordination between sender and receiver. The receiver can easily distinguish garbage from plain text in the decrypted string.

3.3. Random Distribution of Message in Padded Buffer
Random distribution of the message in a buffer is a technique in which the message bytes are distributed throughout a larger buffer. This could be as simple as having the message start at a random position within an augmented plain text buffer. It could also be more sophisticated by having the plain text bytes occur randomly through the augmented plain text buffer. Of course, in both these strategies the sender and receiver must coordinate so that the receiver knows where the plain text bytes occur and where they are.

Random distribution might be especially useful when combined with a special kind of padding in which the padded buffer contains the same number of all the characters used in messages. For example, a set of XML messages might use only the 26 lower case letters, the 10 digits, and a handful of special characters such as ‘<’, ‘>’, ‘=’, etc. If none of these characters occurred more than, say, 4 times in any valid message, then the system might use message buffers 4 times as large as the character set. Before a message was encrypted, the buffer would be filled with a random permutation of 4 copies of the character set. The actual message would then be randomly distributed throughout the buffer. Each time a byte b from the plain text replaced a byte c in the buffer, some other occurrence of c would be replaced by b so that all characters continued to occur the same number of times. The message would then be encrypted and sent to the receiver. This would prevent an attacker from using knowledge about the structure of XML to help crack the code.
4. Conclusion
We have presented a very lightweight encryption method suitable for use on even the simplest of processing elements. The method is also easily implemented directly in hardware. The method is very fast and well suited for applications where memory limitations and real-time constraints are significant factors. We have also presented three additional techniques for increasing the security for applications in which the nature of the messages might provide aid to attackers. Future work includes further study on generating tables with good properties and the relationship of MTM to one-time pad encryption [16].

References
[1] https://csrc.nist.gov/Projects/Lightweight-Cryptography - call by NIST for development of lightweight encryption algorithms.
[2] Manifavas C, Hatzivasilis G, Fysarakis K and Papaefstathiou Y 2016 A survey of lightweight stream ciphers for embedded systems”, Security Comm. Networks 9 1226–46
[3] Buchanan W J, Li S and Asif R 2017 Lightweight cryptography methods Journal of Cyber Security Technology 1(3-4) 187-201
[4] Singh S, Sharma P, Moon S Y, and Park J H 2017 Advanced lightweight encryption algorithms for IoT devices: survey, challenges and solutions Journal of Ambient Intelligence and Humanized Computing doi: 10.1007/s12652-017-0494-4
[5] Biryukov A and Perrin L 2017 State of the art in lightweight symmetric cryptography Available at https://eprint.iacr.org/2017/511.pdf
[6] Hatzivasilis G, Fysarakis K, Papaefstathiou I, and Charalampos C 2018 A review of lightweight block ciphers J Cryptogr Eng 8 141–184
[7] https://en.wikipedia.org/wiki/Chosen-plaintext_attack
[8] Rivest R, Shamir A, and Adleman L A 1978 Method for Obtaining Digital Signatures and Public-Key Cryptosystems Communications of the ACM 21(2) 120–126
[9] https://en.wikipedia.org/wiki/Random_permutation
[10] https://en.wikipedia.org/wiki/Stream_cipher
[11] https://en.wikipedia.org/wiki/Advanced_Encryption_Standard
[12] https://en.wikipedia.org/wiki/Cryptanalysis_of_the_Enigma
[13] https://en.wikipedia.org/wiki/Semantic_security
[14] Shannon C E 1945 A Mathematical Theory of Cryptography Bell System Technical Memo MM 45-110-02
[15] https://en.wikipedia.org/wiki/Chosen-ciphertext_attack
[16] https://en.wikipedia.org/wiki/One-time_pad