Encryption using ad-hoc random algorithm generation

Linus Lee

June 24, 2016

Abstract

Most modern-day symmetric-key block cipher encryption schemes use a base set of key(s) as numerical parameter(s) of mathematical operations to obscure plaintext. However, a different class of symmetric-key encryption schemes could be conceptualized where, rather than an encryption function manipulating the bit-level data in a predictable way belonging to a family of functions, the encryption key is used as a seed to a (pseudo)randomly generated bitwise manipulation function. In this article we introduce a method of generating ad-hoc encryption functions for each instance of encryption taking the key as a seed. Using this concept, we then propose a possible symmetric-key block cipher around it, and observe the implicatinos of such a symmetric-key method through its cryptoanalysis.

Contents

1 Functional encryption

With the growing importance of digital and storage security and threats that attempt to decrypt local storage rather than transport layer data, symmetric encryption is of growing importance. But as the spotlight continues to be on transport layer security and asymmetric ciphers, there has been less focus on symmetric-key ciphers in the last few years than they warrant, in no small part due to the fact that they standard of technologies in this space has been rather stagnant. In this space, rather than iterate or attempt to improve on a current standard, we may look to more experimental concepts on different paradigms of symmetric encryption.

Traditional methods of encryption use a pre-determined algorithm, taking the encryption key and the plaintext as numerical parameters. Indeed, traditional encryption methods are defined by their corresponding families of functions. For example, the Rijndael / AES cipher uses predefined byte-level operations, `SubBytes`, `ShiftRows`, `MixCols`, and `AddRoundKey`, acting on a substitution-permutation matrix [1].

While the use of predefined algorithms simplifies the structure of the encryption scheme, it also makes for a simpler cryptoanalysis of the scheme and creates a predictable algorithm to be analyzed and attacked by an adversary. Moreover, it lends certain uniform mathematical properties to the algorithm that may lead to a universal vulnerability.

To further obfuscate cryptoanalysis and minimize room for cryptographic breaks, we may attempt to create an algorithm without a preestablished set of operations. In other words, we may posit an encryption scheme that creates a new, unique bit-level encryption function at each instance of encryption from the key. Such an instance-specific, ad-hoc cipher algorithm would not only easily elevate the trapdoor-ness of the cipher, but create a complex algorithm for cryptoanalysis and consistent attacks without substantially increasing computation time.

In structure, a symmetric-key algorithm of such a *functional encryption* cipher may be 1) derive a unique, key-specific encryption algorithm from the key or subkeys, 2) encrypt the plaintext using the custom algorithm, and then 3) impose an extra layer of security by applying a second-layer algorithm on top. This is the structure we follow in this investigation of an experimental algorithm, where the main cryptographic algorithm is generated on the fly, as specific and unique to the key.

1.1 Methods of instance-specific algorithm composition (ISAC) using bitwise operations

One of the simplest possible ways of constructing an ad-hoc algorithm generation scheme is by sequentially chaining together a string of a series of elementary bitwise operations like addition, subtraction, and inversion according to a predefined lookup table associating each subkey derived from the key to a specific operation.

Already it is possible to see that even with such a basic construction, given long-enough keys, a ciphertext generated by such an algorithm would be nearly impossible to reverse in polynomial time, let alone easily. To "break" the cipher via either plaintext or key identification, the adversary must recover the entire series of elementary bitwise operations underwent in the entire algorithm, since it is impossible to assume any pattern in the series of sequential bitwise operations executed.

While these "elementary operations" could theoretically be any operation on a bit stream that could be composited in sequence with other operations, bitwise operations have a particular appeal because of their inherent simplicity – meaning they make for a better trapdoor function when used in random sequence, because of their inherently elementary nature, and bitwise operations are inherently fast in any modern computing hardware and low-level programming languages.

However, in compiling a set of possible operations for the creation of a standard encryption scheme, the set of possible operations must be chosen carefully as to allow for several possiblilities. The reason for selection of the particular elementary operations chosen for ISAC will be explained later.

2 Structure of the ISAC encryption scheme

The design of the ISAC encryption algorithm is built with the core goals of *resistance to computational abbreviation* and *lowering computational overhead* in mind. The full ISAC encryption process occurs in two main steps, the first of which is intended to do the bulk of encryption with higher computational overhead and the second of which is intended to provide additional statistical security with far lower computational overhead. Both stages can be hardware accelerated or designed into a dedicated processing core's fabrication.

The algorithm is a block-level encryption algorithm, operating on a 64-

byte block at a time with a key of any $8n$-bit length key[1] In the particular implementation investigated in this article, we have used an encryption scheme with the key length of 256 bits operating in blockchain mode[2] in C for `x86_64` platforms.

The scheme uses a standard CBC (blockchain) implementation with the 64-byte blocks, using `xor` operations between the previous plaintext block and the current block as the current block's plaintext.

2.1 The G-algorithm

The defining feature of ISAC is its on-the-fly composition of an encryption "algorithm". Within the stack and source, this instance-specific encryption algorithm is a bit-level sequence of operations operating altogether on each 64-byte block of the ciphertext at a time, referred to as **G**. **G** is generated as a sequence of randomly ordered, randomly selected, four different simpler elementary bit-flicking operations. These bit-flicking operations each take a single power-of-two parameter, and operate on an n-bit section of a binary stream, where n is the operational argument given in the last six bits of each byte of the key. (The process of the construction of the **G** algorithm from the key will be elaborated further in a later section.) They are[3].

1. **add(n)**: adds 1 bit to every nth byte, e.g. 01101001 $\rightarrow$ 01101010
2. **cycle(n)**: equivalent to a circular rotation of the bits right by a single bit for every nth byte, e.g. 01101001 $\rightarrow$ 10110100
3. **reverse(n)**: reverses the order of the bytes in nth-bytes of the block
4. **split(n)**: splits and juxtaposes a section by even/odd-number byte pairs for every nth-byte

and the entire instance-specific algorithm is generated as a sequence of a compound of these operations. For instance, a particular **G** may take the form `add(4) reverse(2) split(16) add(8) reverse(2) cycle(32) add(4) cycle(8)`. In effect, this can be operated upon the entire binary stream in sequence,

[1]Here, n corresponds to the number of bitwise operations conducted in the **G** step.

[2]The open-sourced code may be found at `https://github.com/thesephist/voyage/`.

[3]here, the nth byte begins count with 1, going up as every $1 + in$ bytes where i is an integer, for brevity in code.

iterating over each block (and ideally in a stateless blockchain manner), overcoming the problem of necessitating a key as long as the encrypted data itself while preserving the same, inherent and security from randomness.

2.1.1 On-the-fly construction of G

The **G** algorithm is constructed as a sequential set of operations of the four elementary bit-flicking operators presented above. In the model investigated here, the construction is derived directly from the 256-bit encryption key. However, a subkey or set of subkeys derived from the main key may also be used here. The only technical requirement here is that the key be a stream of bytes.

G is constructed sequentially from the key. As a consequence, the equivalent operation of creating the algorithm **G** and then implementing it can be performed simultaneously, with the elementary operators acting on a 64-byte block sequentially as the key is being read.

Each byte of the key corresponds to a single elementary operation, resulting in a total of 32 operations for a 256-bit key as we have used here.

For each byte of the key, the first two bits, taken together as an unsigned integer 0 - 3, denotes the *kind* of elementary operator used, where the last 6 bits, taken together as an unsigned integer 0-63, denotes the parameter of the particular operator as it is executed on the block.

2.2 The H round

The **H** algorithm is, by definition, simply any algorithm that juxtaposes the bit-level output of **G** and the encryption key in a way similar to OTP with round keys to create further obfuscation of pattern recognition to any extent. The simplest implementation, would, for example, be a simple `xor`-with-key operation for each bit, where the key has been repeated to match the length of the encrypted output of **G**.

For the purposes of this investigation, the **H** round algorithm employed was a modified version of it, such that after the `xor` step, the ciphertext was bit-circulated right an arbitrary number of bits.

3 Design motivations

Each of the elements in the high-level design of the algorithm, specifically the execution of the `add, cycle, split, reverse` operations, was considered carefully for their role in the larger context of the algorithm before being integrated into the scheme, and each of the four were chosen for a variety of different contributions and effects to the overall security of the main encrypting round **G**.

The same can be said for the construction of the arbitrary juxtaposition operation **H**, where, in the case of the scheme's implementation investigated here, the algorithm is simply an `xor` pass over the output of **G** with respect to a rounded-length key. In more secure implementations, this may be altered.

Let us explore the motivation for each element of the scheme's design in both of the above cases.

3.1 `add` subround

Among the four elements of the **G** round, the `add` operation is the only one that flips bit values; all other operations merely generate permutations of bits. Thus the `add` subround is integral to the round in introducing an additional factor of complexity through (pseudo-)randomly changing certain bit values according to the key.

Although the `add` operation in itself only increments the numerical value of a byte by a unit in a predictable fashion, through permutation of bits and bytes introduced in other rounds and through multiple executions of the step with varying parameters, the operation generates an element of randomness vital to the set of **G** operations.

3.2 `cycle` subround

The `cycle` operation strikes a balance between keeping performance high and effective obfuscation equally high. The operation takes place at the bit-level, performing `RCIRC(1)` for each applicable byte of information. The resultant change in information contrasts sharply with the previous `add` operation – `cycle` introduces a change in the magnitude of the byte-level numbers that is fundamentally different in scale than `add`, because while the former adds a single unit, the latter changes the numerical value of the byte arbitrarily and entirely.

3.3 `reverse` subround

The general effect of the `reverse` operation is the same as those of the `cycle` operation, and the operator functions as a second and juxtaposing way to cause a different kind of large perturbations in byte-level data. In practical applications, `reverse` is implemented through a lookup table.

3.4 `split` subround

The `split` subround is the most complex of the four, and as such, adds the most complex permutations to the plaintext. The previous three subrounds alter byte-level information very regularly. While one alters the byte barely by a couple of bits and the other two merely transform the bit orders, `split` introduces a different kind of bit-level juxtaposition into the mix, and this added complexity adds to the difficulty of cryptoanalysis of the encryption.

3.5 H round

The **H** round in this implementation has two distinct steps: the `xor` and the `rcirc` steps. The main design goal of the **H** round is to add a layer of basic obfuscation so the ciphertext is not malleable (and the possibility of leakage of plaintext information is minimal). This is done in two steps, both of which adds a factor of randomness and chaotic behavior to the plaintext-ciphertext relation.

In the `xor` step, a roundkey is generated from the given key simply by repeating the key until the length of the key is equal to the length of the ciphertext output from **G**. Then this preliminary ciphertext is xor'd with the roundkey.

In the `rcirc` step, the sum of each individual bit in the result of the xor step, times 72, is divided by the ciphertext bitwise length, leaving a remainder. Then the entire preliminary ciphertext is bitwise-rotated by the remainder.

This sequence of step produces a concluding step to the encryption algorithm such that a small change in either the ciphertext or the key will generate a deviation in the final ciphertext that is large beyond any remediation.

Sampled block size	frequencies of occurence[4]	σ
1-bit	102990, 105686	1.35
2-bit	18218, 18723 18373, 19566	1.11
3-bit	6158, 6222, 6044 6084, 6172, 6052 6046, 7142	1.243
4-bit	2473, 2101, 2218 2316, 2150, 2382 2223, 2298, 2093 2185, 2533, 2989 2290, 2449, 2551 2189	1.251

Table 1: Statistical distributions of various blocks' occurences in an ISAC-encrypted binary stream

4 Statistical security

While the complete properties of a supposedly secure encryption scheme ought to be examined through various cryptoanalytic techniques rigorously and mathematically, due to a number of mechanical and time barriers, we neither have the time nor the resources to conduct a comprehensive cryptoanalysis of ISAC. For this reason, a more complete cryptoanalysis will be delegated to a future study, while here, we merely examine certain common adversaries as well as the statistical properties and pseudorandom advantages of various ISAC schemes.

Here we use the term *statistical security* to mean the independence of statistical properties between the plaintext and the generated ciphertext.

As a brief demonstration, a sample of around 18K bytes was encrypted with the given algorithm and randomly generated keys to obtain a sample of an equivalent length of ciphertext. The statistical frequencies of unique bit-sequences appearing in the ciphertext is noted in Table 1, alongside the frequencies of equivalent bit-sequences in a pseudorandomly generated 18KB binary.

In simple terms, the apparently high entropy in the ciphertext signifies the lack of dirrect correspondences between the cipher and the plaintext, and makes it naturally more difficult to search for keys that match the plain-

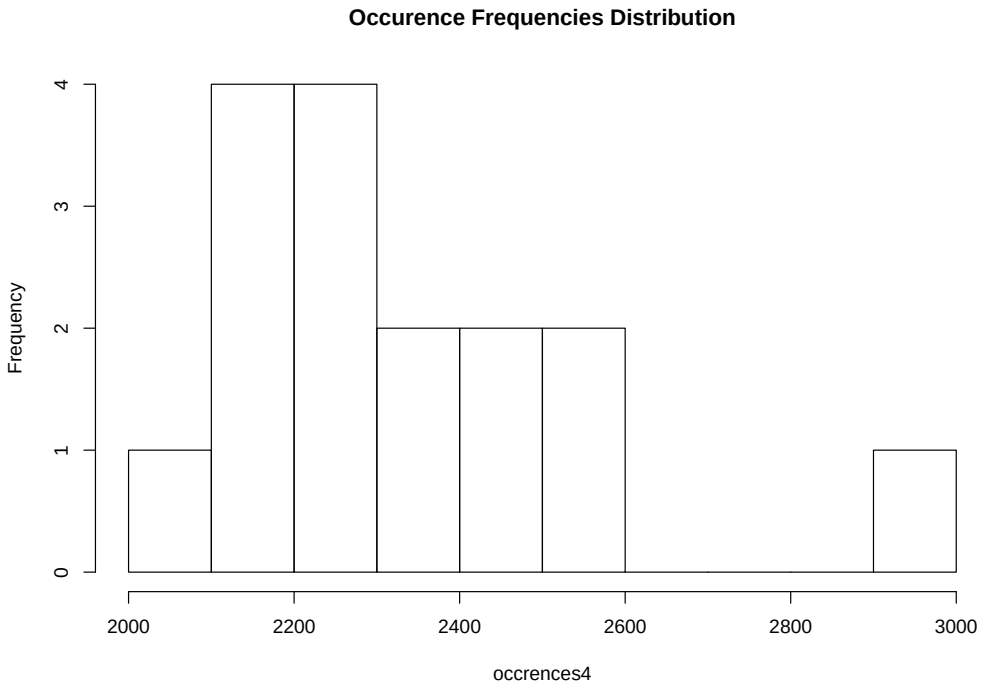

Figure 1: Histogram of occurences of 4-bit binary blocks' occurence frequencies in an ISAC-encrypted binary stream.

text. In essence, the pseudorandom nature of the ciphertext resists faster decryption.

5 Cryptographic security

In this investigation we examine the cryptographic security of ISAC in the following ways. First we will look into ISAC's resistance against traditional non-quantum cryptoanalysis. Then we will discuss various attack vectors into ISAC's construction from known- plaintext, chosen-plaintext, and chosen-ciphertext attack perspectives.

Tranditionally, on algorithms such as DES and AES / Rijndael, cryptoanalysis is performed at the algorithm level. In other words, any attempt at finding vulnerabilities within the algorithm only required using one of the aforementioned common methods to discover properties about the algorithm being used (such as, for example, exploiting the properties of a Feistel network, which is the same for all AES ciphers). The fact that traditional cryptography almost always resorts to some fixed element and fixed order of execution upon those elements meant that a cryptoanalysis on the general cipher was easily a cryptoanalysis on any one specific instance of the cipher.

In stark contrast, ISAC is designed precisely to be resistant to a one-time crytoanalysis. In other words, because 1) multiple different permutations of **G** lead to the same ciphertext from the same plaintext frequently and 2) the precise permutation of the base algorithms inside **G** is unique each run of the ISAC encryption process (assuming distinct keys), a cryptoanalysis against one instance of the cipher is merely an attack on one particular instance.

However, while the core of the ISAC cipher is resistant to any attacks against a nonexistent fixed property of the algorithm, depending on the implementation of the **H** algorithm, a differential cryptoanalysis with chosen plaintexts may leak certain properties about a given plaintext[5].

Conducting a known-plaintext attack (KPA) on ISAC requires...

A chosen-plaintext attack on ISAC is...

A chosen-ciphertext attack on ISAC...

[5]Note that while the chosen-plaintext attack on **H** leaks some information, without resorting to an equivalent of a brute-force search, due to the redudancy in many permutations of the **G** algorithms, it is effectively impossible to recover the plaintext, in part or in entirety, through this method.

6 Nonmalleability

In the construction of ISAC, effort has been put in to harden the algorithm against malleability, or the possibility of modifying ciphertext while preserving the decrypt-ability of the ciphertext. A malleable cipher is exposed to attacks where the adversary is able to change the plaintext in arbitrary ways, even if the key is not recoverable.

ISAC defends against malleability attacks in two ways.

First, by nature of its construction, each bit or pattern in the ciphertext is not traceable to any initial bit or character, as the bit sequence is put out of order and switched around as a consequence of the randomized juxtaposition process in the **G** round.

However, the **G** process by itself fails to completely mask the relation between plain- and ciphertext, as a small change in the plaintext maps to a small change in the ciphertext. Though the bits are not correlated in position, they are correlated in the degree to which the information changes. In order to offset that relation, the **H** round adds a secondary layer of re-hashing the ciphertext into a less correlatable form.

7 Conclusions

ISAC is, more than an attempt to construct a fully working implementation of a cryptographic system that can be integrated into a production environment, a proof-of-concept (POC) for a different way of conceptualizing mapping between cryptographic keys, the algorithms, and the ciphertext.

Rather than the conventional and most frequently seen key-ciphertext relationship where the key is simply a numeric value (or the origin of numeric-value subkey sets, as in AES) as the parameter of a set sequence of subprocesses inside the master cryptographic algorithm, ISAC is a POC of a cryptographic algorithm that directly maps a unique and distinct algorithm to each key, dramatically increasing the complexity of the key-ciphertext and key-algorithm mappiong without significant drawbacks in speed of execution and complexity of the algorithm.

This form of constructing cryptographic algorithms, here called functional cryptography, may provide other avenues of exploration as traditional techniques of cryptography is broken by quantum computing and massively parallel GPU-driven systems. Functional cryptoalgorithms' extremely high

nonlinearity and abundance of plaintext collisions may prove useful against quantum cryptoanalyses.

References

[1] Shafi Goldwasser, Mihir Bellare, July 2008. *Lecture Notes on Cryptography*. MIT OpenCourseware.