



## White Paper

# How to Encrypt a Credit Card Number

- Terence Spies, CTO, Voltage Security, Inc.

## Table of Contents

Overview	3
The Encryption Security Model	3
The Perfect Credit Card Encryption Algorithm	5
Perfect Approximation One: The Prefix Method	6
Perfect Approximation Two: Feistel Networks	8
Conclusion	11

## Overview

The arrival of the PCI Data Security Standards has driven many organizations to search for ways to encrypt CCNs in internal databases. While this can seem like a straightforward application of encryption algorithms, encrypted data typically has a different format from the plaintext, which necessitates changes to database schemas, and reworking business applications so they can recognize the new encrypted data format. Counter-intuitively, it turns out there are well-known cryptographic techniques that encrypt structured data like CCNs without changing format, and which retain the high level of security associated with standard block ciphers like AES. This technical note describes how these techniques (called Format Preserving Encryption algorithms) work, and how they can retain cryptographic security without expanding the data.

## The Encryption Security Model

The goal of encrypting data is taking some plaintext information and rendering it useless to an attacker. That is, an attacker stealing encrypted data should get no more value from the stolen data than if they had just generated random data on their own. All encryption algorithms are imperfect, since they use large but finite length binary numbers as encryption keys and, an attacker who can guess cryptographic keys, can recover the plaintext.

Encryption algorithms are typically classified by how many binary bits are used to construct the cryptographic key. A “128-bit” encryption algorithm will use 128-bit long binary numbers as keys, and an attacker (unless the algorithm has some weakness) will need to examine all, or nearly all of these keys to find the plaintext. The magnitude of this task is apparent once you realize this requires looking at up to 340,282,366,920,938,463,463,374,607,431,768,211,456 keys. If an attacker can look at a billion keys a second, the attacker is looking at up to 10,790,283,070,806,014,188,970 YEARS of effort. 256-bit algorithms make this problem exponentially more problematic. No one realistically expects 128-bit keys to be broken with conventional computers. The main motivation for going to a 256-bit key is speculative and regards quantum computers that might be invented in the future.

Another way to look at encryption and keys is to model encryption as a shuffling process. An encryption function takes in a set of plaintext values, and shuffles each plaintext value to a distinct ciphertext value, with the key dictating exactly how the shuffle is done. For example, the AES algorithm with the key “0000000000000000” produces the following “shuffle table” which shows what a given plaintext value is mapped to: (Note that we are using hexadecimal notation here, which is shorter than writing everything in binary.)

Input	Output
0000 0000 0000 0000 0000 0000 0000 0000	66e9 4bd4 ef8a 2c3b 884c fa59 ca34 2b2e
0000 0000 0000 0000 0000 0000 0000 0001	0143 db63 ee66 bocd ff9f 6991 7680 151e
0000 0000 0000 0000 0000 0000 0000 0002	f6b7 bdd1 caee bab5 7468 3893 c447 5487

If we change the key to “0000000000000001”, we get a completely distinct table:

Input	Output
0000 0000 0000 0000 0000 0000 0000 0000	5c76 002b c720 6560 efe5 50c8 0b8f 12cc
0000 0000 0000 0000 0000 0000 0000 0001	72b4 c1a4 5b14 ec39 a893 456f 2ed1 75a3
0000 0000 0000 0000 0000 0000 0000 0002	ec33 1f5d d1c5 f40e 28ea 541c aec9 13f6

Writing these tables out completely would be impossible; the important idea here is that all encryption algorithms produce tables like this. We can now use this table idea to see how cryptologists look at encryption algorithms. Cryptologists will consider an encryption algorithm strong if the following conditions apply:

- ▶ **Every key must produce a distinct table.** If this isn’t true, then there are multiple keys that are equivalent, and the key size of the algorithm is smaller than stated.
- ▶ **Even given the ability to look at a huge number of table entries of their choice, an attacker cannot tell what key was used to produce that table without essentially trying all keys.** This means that, no matter how much data an attacker has, they cannot take any shortcuts to finding the key. (In the technical literature, these are called “known plaintext”, “known ciphertext”, “chosen plaintext”, or “chosen ciphertext” attacks.)
- ▶ **Even given the ability to look at a huge number of table entries, an attacker cannot predict anything about the relationship between an unknown input and output.** This means that the attacker has to actually possess the key to encrypt or decrypt any data, even partially.

### The Perfect Credit Card Encryption Algorithm

Given these rules, we can now propose a perfect, but impractical way of encrypting credit card numbers. Our “perfect” algorithm will encrypt a credit card number, keep it in credit card number format, and be much harder to break than even using AES-256 encryption. Our “perfect” algorithm will store every credit card number in memory, and then randomly shuffle them until we have produced a table that looks something like this:

Input Credit Card Number	Output Credit Card Number
0000 0000 0000 0001	9456 1238 7623 9012
0000 0000 0000 0002	7123 8829 3128 0023
0000 0000 0000 0003	4128 4328 2811 8239

For our Perfect algorithm, we’ll consider the table itself as the key. This table is extremely large (60,000 GB large, as a matter of fact), but it meets all the criteria set for an encryption algorithm above:

- ▶ **Every key must produce a distinct table.** Every shuffle is, by definition, a distinct table. Note that the Perfect algorithm is much stronger than even AES encryption. While there are  $2^{128}$  possible AES keys, there are  $10^{16}!$  possible Perfect tables. (The ! symbol here refers to the factorial function. To calculate the factorial of n, multiply  $1 \times 2 \times 3 \times \dots$  up to n. The factorial of  $10^{16}$  is not just a huge number, it is inconceivably huge. This number would have over 80,000,000,000,000,000 digits if written out.)
- ▶ **Even given the ability to look at a huge number of table entries of their choice, an attacker cannot tell what key was used to produce that table without trying all keys.** Since we shuffled the table randomly, an attacker cannot predict the rest of the table (which is the key) from what they have seen previously.
- ▶ **Even given the ability to look at a huge number of table entries, an attacker t predict anything about the relationship between an unknown input and output.** Again, because the shuffle is random, the Perfect algorithm has destroyed any relationship between the input and the output.

An attacker who gets access to a database encrypted with the Perfect algorithm, is confronted with data as valuable as data that they could have generated themselves, by randomly making up credit card numbers. Now the question becomes how we can approximate this Perfect function. What we would like is an algorithm that works like the Perfect algorithm, but is practical to implement, and is at least as hard to break as some existing cipher like AES-128. This question is precisely what motivated Black and Rogaway’s research paper “Encryption in Arbitrary Finite Domains”, which is where we’ll find the techniques that we’ll use.

### Perfect Approximation One: The Prefix Method

To find a good approximation for the Perfect method, we'll start with a smaller, simpler problem, and then look at techniques that will work for larger cases. The case we'll start with (and compare with AES for good measure) is encrypting a field that contains two decimal digits.

How good is the Perfect method in this case? It would seem that shuffling a table with 100 elements should be weaker than AES, since it produces outputs that are very large in comparison. It turns out that, even with just two digits, the Perfect method can produce  $9 \times 10^{157}$  tables.

This is equivalent to 364 binary bits of key, which is stronger than even AES-256, so the strategy of approximating the Perfect method is still a potentially secure technique.

It turns out that for cases this small, we can essentially use AES directly to shuffle a table, then use the table to encrypt and decrypt. Instead of storing the table, we can now just store the AES key used to create the table.

The prefix method creates the table using the following algorithm:

Input	AES Output
0	ec331f5dd1c5f40e28ea541caec913f6
1	932c6dbf69255cf13edcdb72233acea3
2	6d5c3e022e5a6f7be663b9e69bcea443
3	e013d7f4fa7abd93a7b85db9cfff9b14
4	foa2a65d245dd6199dc70951c2478b65
5	e7f1b82cad53a648798945b34efeff2
6	fb759742d93ac3coca761d98de09a833
7	df9c0130ac77e7c72c997f587b46dbe0
8	a59981cf84c750e4e82b5b8ebfaf75c6
9	a7bd772fabd574bac462273afb906e9a

- ▶ Write down each possible input value, and encrypt it with AES
- ▶ Sort the table using the AES output as the sort key.
- ▶ The order of the input values in the is now the output column of the table.

We create an example table for 10 items using a random AES key to illustrate how the algorithm works. This encrypts a single digit, but the double-digit case works the same way, with a larger table. Here's the first step, which is a table consisting of the possible inputs and the resulting AES encryption outputs:

(Notice that we could use this table to encrypt our values with AES, but now our ciphertexts are much larger than our inputs. Note also, that the extra bits AES has

Input	AES Output
2	6d5c3e022e5a6f7be663b9e69bcea443
1	932c6dbf69255cf13edcdb72233acea3
8	a59981cf84c750e4e82b5b8ebfaf75c6
9	a7bd772fabd574bac462273afb906e9a
7	df9c0130ac77e7c72c997f587b46dbe0
3	e013d7f4fa7abd93a7b85db9cfff9b14
5	e7f1b82cad53a648798945b34efeff2
0	ec331f5dd1c5f40e28ea541caec913f6
4	foa2a65d245dd6199dc70951c2478b65
6	fb759742d93ac3coca761d98de09a833

produced are not really buying us any security here. If the attacker knows a value from 0-9 is being encrypted, the fact that the output is from a large range doesn't

Input	Output
0	2
1	1
2	8
3	9
4	7
5	3
6	5
7	0
8	4
9	6

make the problem of breaking the cipher any harder. In fact, in some cases, it can make the situation worse. If we encrypt the values from 0-9 by padding them with zeros to fill the 128 bit AES input, then an attacker can take a single value pair from the table and try AES keys. If the attacker then finds a key, where the output value decrypts to all zeros along with the input value, there is an extremely high probability he has found the key used to produce the entire table.)

To continue, we now sort the table by the AES output values, producing a new ordering of the input values.

We can now discard the AES values, and use the new ordering of the input values as the output values of the encryption table:

We now have a small encryption function with some remarkable properties. Let's say that we give the attacker the entire table, except for the last two entries. We now ask the attacker what value the input 8 maps to. The attacker has to choose between the remaining numbers, 4 and 6. It turns out we can prove that, unless they can break AES or have the key used to generate the table, they have no advantage over a purely random coin flip to get this right. We can also show that an attacker, without access to the table, cannot derive any information from pure ciphertext about the key or the plaintext values, except that two identical ciphertexts correspond to identical plaintexts. This same set of properties would be the case for AES.

Through this proof (which is in the Black and Rogaway paper), we can show we now have a method that lets us produce small encryption functions using an existing cipher, which lets us reduce the problem of breaking that function to breaking an existing encryption function. The problem is, it's still impractical for tables that are too large to hold in memory, so we still can't encrypt credit card numbers. We need a more sophisticated method.

### Perfect Approximation Two: Feistel Networks

In the early 1970s, Horst Feistel invented a general construction for building ciphers called the "Feistel Network". This construction was used as the basis for a number of ciphers, including DES and Triple-DES. It has been widely studied, and two researchers, Michael Luby and Charlie Rackoff, published an important result in 1988. They showed that this technique can be used to build a cipher from a one-way function, where breaking the derived cipher (within certain constraints) is as hard as breaking the function it is based on. Specifically, it can be used to build an arbitrarily sized cipher from another fixed size cipher, which is exactly what we are looking for.

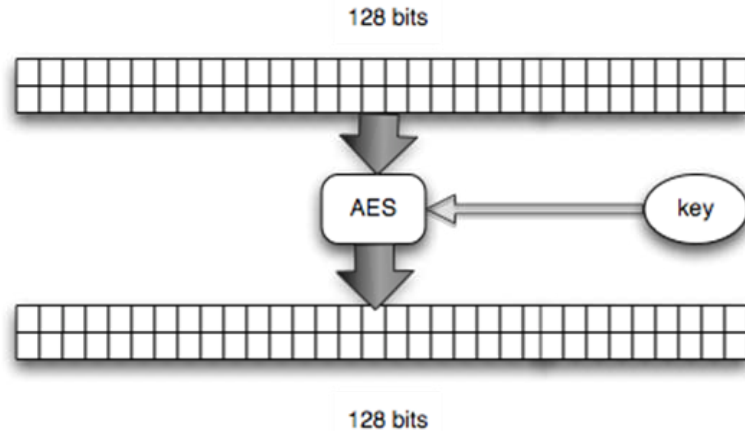
The Feistel network is a simple and elegant construction. It is based on the following simple algorithm:

- ▶ Split the input into a right half and a left half. (If we are encrypting a credit card number, this can be as simple as taking the first eight digits as the left half, and the last eight digits as the right half.)
- ▶ The left half of the output is just the right half of the input.
- ▶ Add the output of the one-way function run on the right half of the output to the left half of the input. This is the right half of the output.

We can express this more concisely in mathematical notation. If R and L are our input, R' and L' are the output, and F(x, key) is our one-way function, the Feistel round is computed as:

$$L' = R$$

$$R' = L + F(R, \text{key})$$



This round function, while very simple, has some interesting properties. It only encrypts half of the input, but observe how we can very simply decrypt by doing the following:

$$R = L'$$

$$L = R' - F(L', \text{key})$$

Now, if we run this round function multiple times, it will encrypt the entire input. The first round will encrypt the right half, and the second round will encrypt left half, since it was swapped during the first round function. We can run these rounds as many times as we like, and still decrypt by simply reversing the steps. But, observe that we just need the same one-way function for encryption and decryption. This means we need a strong one-way function, but don't need to reverse that function. This is the vital property that lets us use AES to do this encryption.

The reason that we can't simply use AES to encrypt, and then shorten the AES output, is AES is a block function. It requires 128 bits of input, and produces 128 bits of output. To decrypt, all 128 bits of ciphertext must be supplied:

If we don't need to decrypt, on the other hand, AES works quite well as a one-way function. (There are number of cryptographic papers, which prove that taking a subset of the output of a strong block cipher yields a strong one-way function.) We can build our one way function, by using our right-hand-side data as in the input to AES, and truncating the output to get as much data as we need, to do the add to the left-hand-side.

Our round function now looks like this:

$$L' = R$$

$$R' = L + \text{AES-encrypt}(R, \text{key})$$

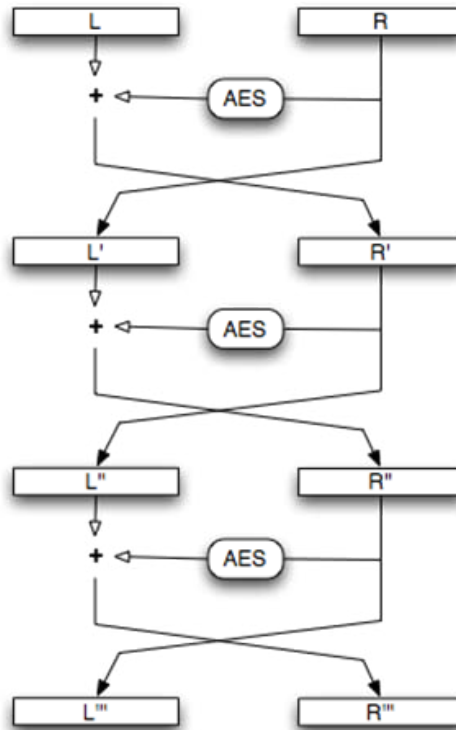
To actually encrypt, we'll run that round function a number of times. We can still decrypt by doing:

$$R = L'$$

$$L = R' - \text{AES-encrypt}(L', \text{key})$$

as many times as we ran the encryption round function. The resulting cipher, if run for three rounds looks something like this:

The only downside here, from a performance standpoint, is we need to run AES one time for each round. Running eight rounds will require eight invocations of AES, for example. AES, on most modern processors, runs at approximately 200 million invocations a second, so for most applications, this will not be a concern. Our only



remaining question is: can we argue that this function is a good approximation of Perfect, or at least as good an approximation as AES is?

The proofs, in the Luby-Rackoff paper and the Black-Rogaway paper, establish how strong this construction is in the case where three or four rounds are used. What they show is: if you give an attacker part of a shuffle table from a Feistel network,

and part of a table from the Perfect algorithm, even with infinite computational resources, they cannot tell which table is the Perfect table, and which the Feistel table is. (With the proviso that an attacker who breaks the one-way function can tell.) If you cannot tell a Perfect table from a Feistel table, you cannot derive any other information about the Feistel table either. If you could, you could use that information to distinguish the Feistel table from the Perfect table.

Jacques Patarin improved this result in a series of papers published around 2004. What he showed is if you run a Feistel network at six or more rounds, you are essentially as strong against both imaginary (infinitely powerful) and realistic (bounded computational power) opponents. The other reassuring property of Feistel networks is that they are a very old—and extremely well studied—cryptographic construction. These papers are a small fraction of the academic papers published on this subject. The fact that DES is based on a Feistel network, has exposed this construction to decades of intense cryptographic scrutiny.

The net result is we can use this construction as the basis of a Format Preserving Encryption algorithm, which allows us to show that the easiest path an attacker has in breaking the resulting encryption, is to break the underlying cipher, which is AES.

## Conclusion

While it seems counterintuitive to think that a cipher, with a small output, can be as safe as a cipher with a large output, in certain cases this is true. Ciphers like AES are designed to encrypt a large amount of data very efficiently. When encrypting large files or disk sectors, the large block size of AES is an important asset. On the other hand, it hinders the use of encryption for small data items. If we purpose-build a cipher for specific data types, we get extremely good security results for ciphers that do not require the data expansion imposed by “vanilla” AES encryption.

## About Voltage Security

Voltage Security, Inc., an enterprise security company, is the global leader in information encryption. Voltage solutions—based on next generation cryptography— provide encryption that just works for protecting valuable, regulated and sensitive information persistently, based on policy. Voltage delivers power, simplicity and the lowest total cost of ownership in the industry through the use of award-winning Voltage Identity-Based Encryption™ (IBE) and its latest innovation: Format-Preserving Encryption (FPE).

Voltage Security offerings include Voltage SecureMail™, Voltage SecureData™, Voltage SecureFile™ and the Voltage Security Network™ (VSN), a Security as a Service (SaaS) solution for the extended business network.

For more information, please visit [www.voltage.com](http://www.voltage.com).

Copyright © 2008 Voltage Security, Inc. All rights reserved. All information in this document is subject to change without notice. This document is provided for informational purposes only and Voltage Security, Inc. makes no warranties, either express or implied, in this document.

Voltage Identity-Based Encryption, Voltage SecureMail, Voltage SecureFile, Voltage SecureData, Voltage Data Protection System and the Voltage Security Network (VSN), are trademarks of Voltage Security, Inc. All other company and product names may be trademarks of their respective owners.