# Approaching the Perfect Cipher by Trespassing the block size confinement – The Polymorphic Giant Block Encryption Algorithm

C. B. Roellgen, PMC Ciphers, Inc.

28.07.2010

*Abstract*

*Maximization of the attack security of a cipher directly entails maximization of the avalanche effect, bit independence and nonlinearity. Conventional synchronous stream ciphers as simple combiner-type algorithms exhibit no avalanche at all and popular block ciphers as being heavily promoted by Secret Services feature tiny block sizes compared with the size of today's user data packets and text- or multimedia files. With increasing block size the number of bytes to pad increases as well, especially for data that is exchanged through the internet with comparably small MTU sizes.*
*A new class of Polymorphic Encryption Algorithms breaks the block size confinement of conventional block ciphers. It further satisfies the Strict Avalanche Criterion in an unprecedented way for a wide range of block sizes while keeping the original plaintext size. This new method enables for encryption of blocks with variable sizes like TCP/UDP data packets without the need for padding plaintext blocks.*

*Key words: polymorphic, encryption, cipher, variable, block, size, stream, synchronous, self-synchronizing, combiner-type, substitution, permutation, key, plaintext, ciphertext, confusion, diffusion, strict avalanche criterion, SAC, completeness, bit independence, nonlinearity, cipher block chaining, CBC, electronic codebook, ECB, initialization vector, AES,* Rijndael*, DES, Luby, Rackoff, balanced, unbalanced, Feistel network, hash, compression, combined secrecy system, Maximum Transmission Unit, MTU, hash, compression, pseudorandom number generator, PRNG.*

## 1. Introduction

In cryptography, the two main properties of secure cipher, as identified by C.E. Shannon [1] from an information theoretical point of view are "confusion" and "diffusion". His theory is the basis of modern cryptography.

"Confusion" refers to making the relationship between key and ciphertext as complex as possible while "diffusion" is a property that leads to best possible distribution of any redundancy in the statistics of the plaintext in the statistics of the ciphertext. The better the non-uniformity in the distribution of individual neighbouring bit patterns in the plaintext is redistributed into the non-uniformity in the distribution of a much larger bit pattern of the ciphertext, the better the transformation. In a cipher with good diffusion, the ciphertext should change in a pseudorandom manner, if only one bit in the plaintext is changed.

In synchronous stream ciphers a keystream is combined directly with the plaintext using the *exclusive or* operation (XOR). In contrast to self-synchronizing stream ciphers, the binary additive mode of operation yields almost no diffusion. In self-synchronizing stream ciphers are several of the previous $n$ ciphertext digits used to bias the keystream, which yields at least a better transformation.

While stream ciphers operate on a small number of individual digits one at a time do block ciphers transform a comparably larger number of digits at a time. The primary advantage of transforming a large number of bits at a time is when an input is changed slightly by flipping a single bit, the output changes significantly. If half the output bits flip, the quality of the block cipher is very good. A.F. Webster, S.E. Tavares [3] define this so-called avalanche effect as follows:
For a given transformation to exhibit the avalanche effect, an average of one half of the output bits should change whenever a single input bit is complemented. In order to determine whether a given $m$ x $n$ (m input bits and n output bits) function $f$ satisfies this requirement, the $2^m$ plaintext vectors must be divided into $2^{m-1}$ pairs, $X$ and $X_i$ , such that $X$ and $X_i$ differ only in bit $i$. Then the $2^{m-1}$ exclusive-or sums

$$V_i = f(x) \oplus f(x_i)$$

must be calculated. These exclusive-or sums will be referred to as avalanche vectors, each of which contains *n* bits, or avalanche variables.
If this procedure is repeated for all i such that $1 < i < m$, and one half of the avalanche variables are equal to $1$ for each $i$, then the function $f$ has good avalanche effect.

The following example demonstrates the effect graphically. The photo below has been reduced to 64 grayscale levels to ease the demonstration.


*Figure 1: Original image* *Detail*

AES with its 16 byte block length used in ECB clearly reveals all areas spanning only 4x4 pixels with a similar color. ECB mode will surely not be used by anyone who implements AES or a similar cipher. Counter mode or CBC mode solves this deficiency in part.


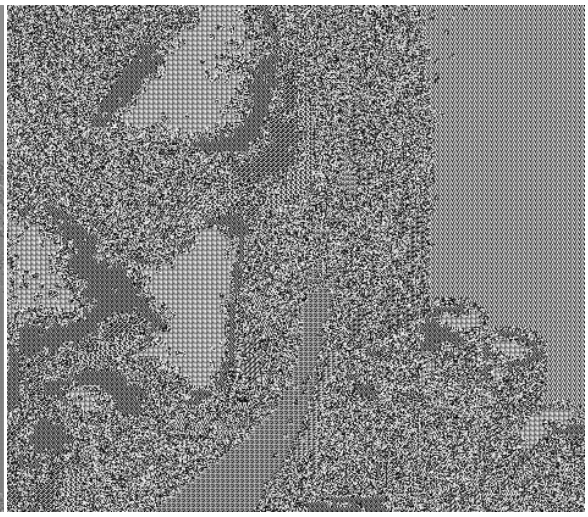

*Figure 2: AES encrypted image (16 byte blocks)* *Detail*

Even a plain 1024 bit block cipher in ECB mode performs visibly better than AES in ECB mode. Adjacent 8x16 pixel wide areas with identical content are encrypted in an identical way which yields the same bit pattern. The mapping is much better, but it is still possible o draw conclusions about the plaintext.

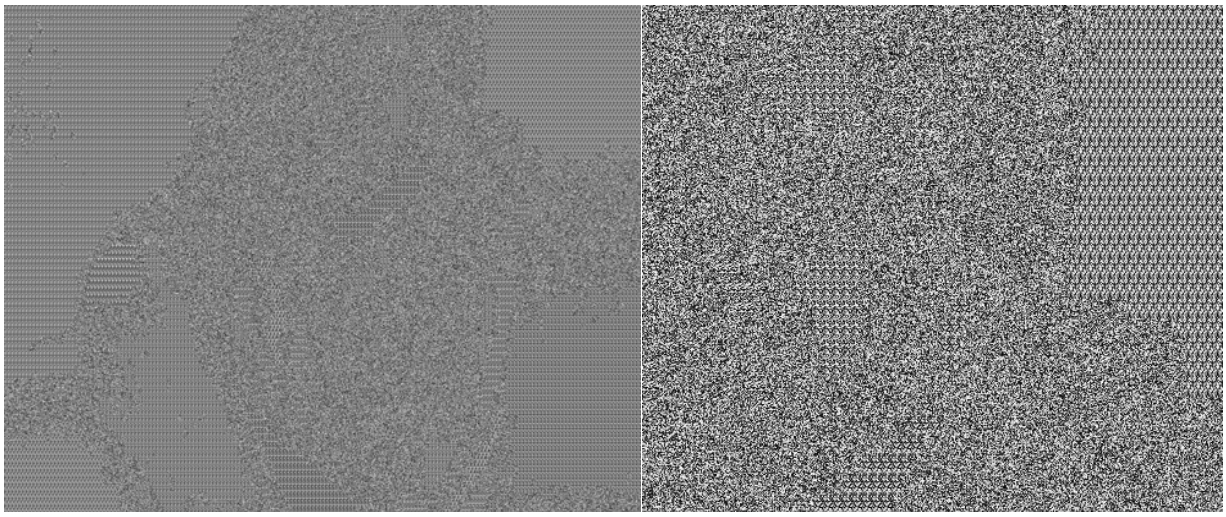

*Figure 3: PMC encrypted image (128 byte blocks)  Detail*

A block cipher featuring a block size that equals the size of this specific sample photograph undoubtedly yields the best result as each and every data bit depends on each other data bit. The cipher used to encrypt the entire photo is as well operating in ECB mode. There's no need to use CBC or any other suboptimal cipher mode as the block size is the size the image file.

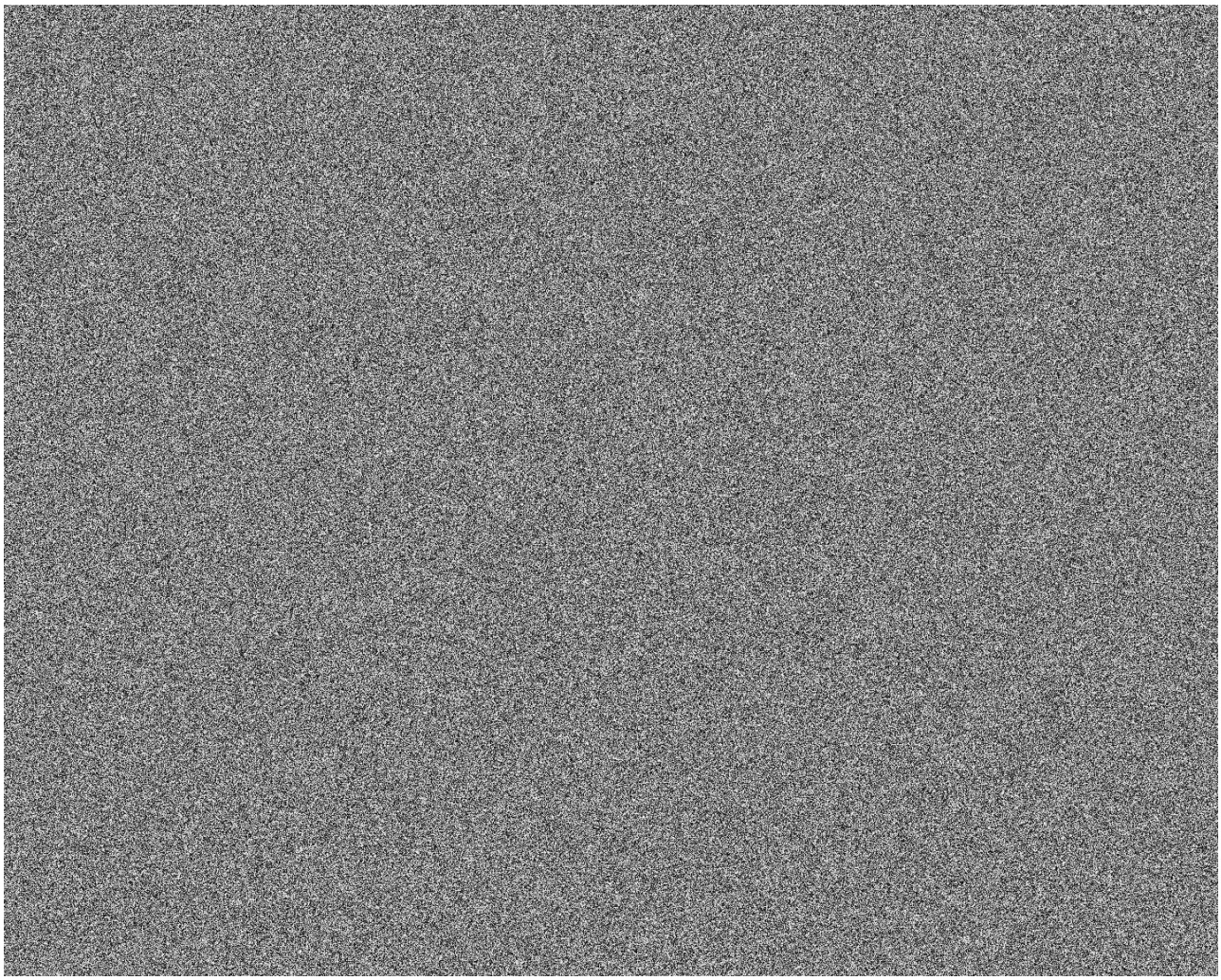

*Figure 4: Encrypted image using a Polymorphic Block Cipher with three unbalanced Luby-Rackoff rounds and 1.6Mb block size!*

When cryptographers claim to have taken the avalanche criterion into account, they usually don't (want to) realize that this is mathematically impossible with block sizes around 128 or 256 bit as the messages that are encrypted with those ciphers are typically much longer. Even typical TCP or UDP packet sizes of only 1Kbyte equal 64 times 16 bytes (128 bits = 16 bytes). The example of an IPv4 packet clearly demonstrates that with only 128 bit block length, almost the entire header of all IP packets with identical length is encrypted into the same ciphertext. If the data area does not contain any block counter, even more static ciphertext will be created. The mapping would be much better if block length of the cipher would at least span up to the first few bytes of the payload. IPv6 headers are even more critical. A cipher with fixed block length of 512 bit would be required to yield a relatively useful mapping! **At this point readers it must be pointed out that "popular" ciphers are always those that have been certified by authorities whose job mainly consists of gathering intelligence. There is a clear conflict of interests for these government organizations. These professionals clearly know about the blatant deficiencies of the encryption algorithms that they certify.**

| bit offset | 0–3 | 4–7 | 8–15 | 16–18 | 19–31 |
|---|---|---|---|---|---|
| 0 | version | header length | differentiated services / type of service | total length of packet | |
| 32 | identification | | | flags | fragment offset |
| 64 | time to live | | protocol | header checksum | |
| 96 | source IP address | | | | |
| 128 | destination IP address | | | | |
| 160 | data (if there are no options – otherwise options are inserted before data) | | | | |

*Table 1: IPv4 packet*

If block size was variable, even between small block lengths such as 256 or 1024 bit, an opponent would at least find it more difficult to break such a variable cipher as he or she would not even know which piece of information belongs to the very first block or the second, third or fourth.
Variability of the cipher per se increases attack security. As an example, an opponent may have the choice between AES Rijndael, Twofish, Magenta and RC6. That's an extra 2 bit key information making life of an opponent four times more difficult without costing the user of a variable encryption algorithm more than a simple selection operation and a few kilobytes of hard disk space, RAM or ROM. By compiling an encryption algorithm, a much larger number of different ciphers become equally probable. I've called this fundamental approach "Polymorphic Encryption" in 1999. This paper describes a much more powerful extension of this concept.

Although in practice difficult to realize, it is without any doubt desirable that block sizes are neither fixed nor limited to a small number of digits. One might argue that cipher block chaining (CBC), a popular mode of operation for a block cipher where each block of plaintext is *exclusive ored* with the previous ciphertext block before being encrypted, provides the required security because each ciphertext block is dependent on all previous plaintext blocks. The very first block is although unfortunately only protected with the initialization vector (IV) and has absolutely no relation with the second, third or any of the following blocks.

While variable, as well as giant block sizes are desirable, a number of additional criteria must be met by a truly secure cipher:

| Design goal | Polymorphic Giant Block Size Cipher | Conventional Ciphers |
|---|---|---|
| Large and variable block size | **Block size is only limited by the resources of the target computer(s). Target systems should run at 500MHz or higher and more than 10Mbyte free RAM should be available. The Strict Avalanche Criterion is thus met perfectly.** | **Not supported at all. Ciphers like AES need little more than 1Kbyte of machine code and a microcontroller typically used in cheap smart cards and washing machines (approx. 20.000 transistors) to run. It is conceivable that such conventional ciphers could have been hardened against all kinds of attacks if more complex implement-tations would have been the target.** |
| No padding to reach block granularity shall be necessary | **Block size is totally variable and blocks keep their length => no padding required, which results in no information being transmitted in vein.** | **DES: 8 byte block granularity,**<br>**AES: 16 byte block granularity**<br>**⇨ Padding required**<br>**A 2048 bit conventional block cipher would require padding to 256 byte blocks** |

| | | |
|---|---|---|
| | | **resulting in dramatic increase in data traffic if used for the encryption of TCP or UDP data packets.** |
| Partitioning of extremely big blocks at arbitrary position | **Blocks that are too big to handle are truncated into sub-blocks with block sizes that are determined by the key as well as the length of the original block.** | **Not supported at all. AES, DES and all other well-known block ciphers feature fixed block sizes.** |
| Resistance against all known attacks | **Due to its variable nature are Polymorphic Ciphers not susceptible to typical attacks that target specific characteristics and/or known weaknesses of fixed ciphers. Brute Force is although applicable to any cipher.** | **AES can be broken easily by DPA (Differential Power Attack) on small microprocessors and microcontrollers [10].** |
| Resistance to future attacks that may cut effective key size by ½ or even 2/3 | **Cutting of effective key size by ¾ would result in still extremely high complexity of $O(2^{256})$ or higher, which is regarded as totally safe for the next trillion years.** | **Cutting of effective key size by ½ results in an extremely low complexity of $2^{64}$. The cipher would be regarded as being broken. [7]** |
| Extremely long key setup time | **> 100ms on a modern microprocessor make comparably short keys safe against Brute Force attacks conducted on a few machines. Extremely long key setup time increases energy consumption multiplied by the time needed for Brute Force by factor 2.000.000.** | **<1µs help attackers to try each and every password combination. This is highly dangerous if short passwords are being used to protect data.** |
| Platform independence | Runs on any 32 or 64 bit microprocessor or microcontroller | Runs on any 8-, 16-, 32- and 64 bit microprocessor and microcontroller |
| Polymorphism and data dependent selection of functions | **The cipher is not only completely variable, but also is the block size huge and unpredictable if truncation is performed. No static weakness is exhibited.** | **Classic ciphers are static and can thus be thoroughly reverse-engineered and analyzed. Cryptanalysis of a mechanism that does always exactly the same is somewhat easier than for a mechanism that never executes the same operation twice.** |
| Use of large amounts of resources | **1 Mbit internal state requires at least approx. 8 million transistor equivalents to run. This alone makes Brute Force Attack more difficult and much more expensive compared with conventional ciphers.** | Less than 50.000 transistor functions are required to build an AES block. **Approx. 1.000.000 AES blocks can run in parallel on an 8'' wafer to try and break a code using Brute Force.** |
| Attacks need to be expensive for an attacker | **The proposed cipher requires a lot of resources and extremely much time for key setup, an attacker requires a "time x resources product" of approx. 200.000 times compared with AES Rijndael when using keys with a similar length.** | **Trying different AES keys requires 50.000 transistor equivalents and less than 1µs. This isn't really all that much. This is a REAL weakness.** |
| High speed | **1500 Mbit/s on an Intel Core i7 950 (3.06GHz) (64 bit C++ code, 1024 byte block length)** | 1000 Mbit/s on an Intel Core Core i7 950 (3.06GHz) (64 bit C++ code) |
| Proven security | **Three round Luby Rackoff features proven security (the mathematical proof follows below); polymorphic encryption is increasingly popular among experts but it's probably impossible to prove security of the entire cipher.** | **Security is not proven. Extensive peer review indicates that the cipher could be broken in the future:**<br>**For 128-bit Rijndael, the problem of recovering the secret key from one single plaintext can be written as a system of 8000 quadratic equations with 1600 binary unknowns. [8]**<br>**Recently has a new related-key boomerang attack on the full AES-192 and the full AES-256 been found by . Biryukov and Khovratovich [9]. A 256 bit key is reduced to a 119bit key when using AES-256. The attack is not applicable to 128 bit keys.** |

*Table 2: Design goals*

## 2. The cipher

The Polymorphic Giant Block Cipher features a provably secure structure with key dependence in all variable parts of the structure. The enciphering operation is further dependent on the block size that can vary greatly. The cipher has the tendency to take advantage of big blocks whenever this is possible. The cipher is a substitution-permutation network operating on a minimum of 128 bit words (16 bytes) and a maximum of bits allowed by the target application and target platform, thus giving an almost arbitrary block size. Block size can thus easily exceed 1Mbyte on a commercial personal computer! The default setting for the maximum block size of the demonstration implementation is 128 kbyte in order not to exceed the cache size of the secondary cache of modern 64 bit microprocessors.

All values used in the cipher are represented as bitstreams. The indices of the bits are counted from 0 to 7 in one 8-bit word, 0 to 63 in 63-bit words, and so on.

For internal computation, all values are represented in little-endian, where the first word (word 0) is the least significant word, and the last word is the most significant, and where bit 0 is the least significant bit of word 0. Externally, we write each block as a plain 64-bit hex number. The cipher encrypts a 128-bit plaintext $P$ to a 128-bit ciphertext $C$ under the control of an internal state derived from the key $K$. The user key length is in principle variable, but should ideally be longer than 512 bit and shorter than 6144 bit. Short keys are mapped to full-length keys (depending on the implementation) of 6144 bits. This mapping is designed to map every short key to a full-length key, with no two short keys being equivalent.

There are no restrictions on the keyspace.

The proposed encryption is $E = DM \circ F_1 \circ F_2 \circ F_3$ , where $DM$ is a decorrelation module and $F_i$ is one Luby-Rackoff (Feistel) round with a keyed pseudorandom function as round function.

The cipher itself consists of:

- key setup that is required only once. The process step creates the internal state for all confusion sequence generators and SP-networks internal to the cipher. Key setup is intentionally computationally costly and thus time-consuming. The demo project can be compiled with a fast key setup routine for testing;
- an initial permutation $IP$;
- partitioning of big, as well as small blocks into slices of different size in a keyed operation;
- Execution of a first Luby-Rackoff round [5] with a long left binary string and a short right binary string;
- Execution of a second Luby-Rackoff round with a short left binary string and a long right binary string;
- Execution of a third Luby-Rackoff round with a long left binary string and a short right binary string;

For decryption, all steps except for the initial key setup are executed in reverse order.

While conventional block ciphers with large but fixed block lengths of e.g. 8192 bit suffer from their coarse granularity, this is not the case for the proposed cipher. If the task was to encrypt 1100 bytes of UDP traffic using an 8192 bit block cipher, the designer of a communication system using this conventional cipher would have only one choice:
As 1100 byte can only be encrypted in 1024 byte chunks ( due to the 8192 bit block length), the data would have to be transmitted in two separate data packets with 1024 byte length. The first packet would contain the first 1024 bytes, while the second packet would contain the remaining 76 bytes + 948 bytes of totally useless padding information. It is not even possible to send all 2048 byte in one data packet simply because MTU size for Ethernet is 1500 byte and for PPPoE it is even less or equal 1492 byte!
The proposed cipher although encrypts the entire 1100 byte in one packet without changing the size of the packet.

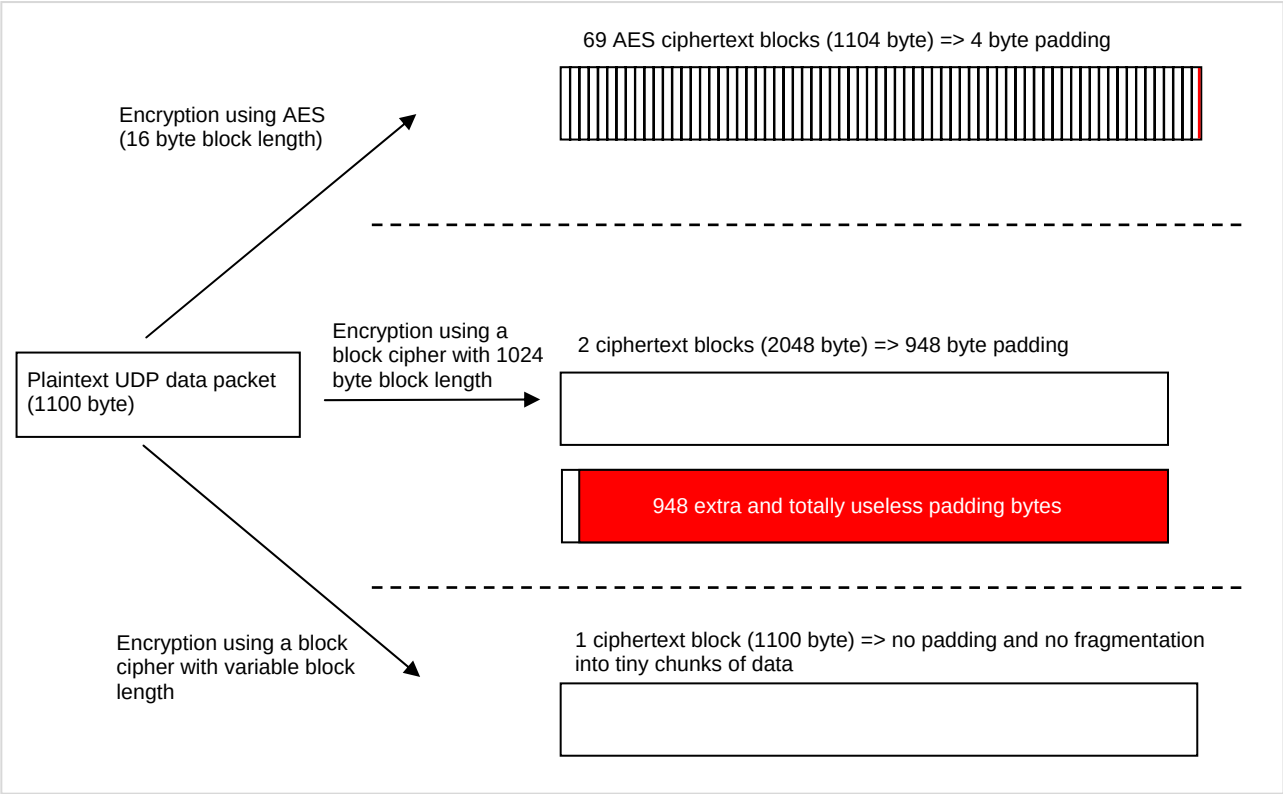


*Figure 5: Encryption of a UDP data packet with block ciphers featuring different lengths*

This example clearly demonstrates one of the most decisive advantages of the proposed class of ciphers – block ciphers featuring large block lengths must be designed to avoid block granularity. Encryption of remaining data could very well be performed by using a cipher with a smaller block size, but it is definitely favourable to encrypt the entire block at once. The figure above further reveals that typical AES-encrypted IP data packets are highly fragmented!

### 2.1 Key Setup

During the key setup phase is the key expanded for use by all sub-functions that require keying. This applies to:

- Confusion sequence generators for the Initial Permutation $IP$ step,
- Shared (and constant) Internal State of the round functions,
- Initial Internal State of the round functions.

Fast Polymorphic PRNG functions can be designed to require an enormous amount of random access memory – for the Internal State as well as for the polymorphic sequence. It is desirable that this pseudorandom data, which is derived from the key, is computed in a lengthy, irreducible number of operations. Another design goal is the irreducible usage of as much RAM as appropriate for the class of target computers the final application program running the cipher actually runs on. For PCs, RAM usage of 1 to 10 Megabytes is very well tolerated by users. Internal State of such size forces an opponent who tries different and possibly likely keys to invest in a lot of chip space and in the electric power to operate 8 .. 80 million transistors. Computing and loading 10 Megabytes of pseudorandom data requires at least 1 million clock cycles. This compares with only 52 bytes of Internal State for the AES Rijndael algorithm that is computed within less than 1000 machine instructions (less than 500 machine instructions on many 32 bit microprocessors). The AES algorithm can be implemented in just a little more than 1Kbyte of machine code and approx. 20.000 transistors, if a very basic CPU is used as target.

*Figure 6: Beautiful Marisa holding a brand new 8'' silicon wafer with 45nm test structures in her hand*

If speed is the target, then less than 50.000 transistor functions are required to build an AES block. Approx. 1.000.000 AES blocks can run in parallel on an 8'' wafer to try and break a code using Brute Force. DES was broken by applying this method in 1998.

### 2.2 Initial permutation *IP*

Initial permutation $IP$ is executed prior to any other operation when encrypting data. For decryption this operation is the final step.
The initial (or final) permutation has the cryptographic functionality to distribute plaintext evenly over the entire block of the global SP-network of the cipher. This decorrelation step shall discourage attackers to apply differential- or differential-linear cryptanalysis to analyze the following round functions.

In [6], Naor and Reingold propose an encryption $E = DM_2 \circ F_2 \circ F_1 \circ DM_1$ where $DM_i$ is a decorrelation module and $F_i$ is one Feistel round with a keyed pseudorandom function as round function. The result is a secure block cipher that cannot be distinguished from a random permutation using chosen plaintext/ciphertext attacks.
In this paper the proposed encryption is $E = DM \circ F_1 \circ F_2 \circ F_3$ , where $DM$ is a decorrelation module and $F_i$ is one Luby-Rackoff (Feistel) round with a keyed pseudorandom function as round function. The advantage of three-round Luby-Rackoff [5] is that security is proven. As the idea behind Luby-Rackoff (Feistel) is stressed for the proposed design (left and right bit strings differ in length – unbalanced Feistel network), usefulness of a decorrelation module is indicated.

Execution time of the simple operation below is *O(n)* because a precomputed table is used in order to speed up execution:

$$ict[i] = p\big[iptable[i]\big]; \qquad \text{for encryption}$$

$ict[iptable[i]] = p[i];$ for decryption

with

| | | |
|---|---|---|
| $ict$ | : | Intermediate ciphertext array (containing permuted plaintext) |
| $iptable$ | : | Precomputed permutation table. Precomputation is performed during key setup |
| $p$ | : | Plaintext array |
| $i$ | : | Index |

$iptable$ is ideally a very big array of 64 bit (or larger) integers, so that e.g. an array sized 65536 entries spans 4Mbit (524288 bytes). This simple method has although the decisive disadvantage that a large number of tables with different sizes are required to enable for variable block sizes.

Another way to perform the initial permutation is much more flexible. A special binary tree that is created during initial key setup pointing to a set of permutation tables in the bottom tree elements is so versatile that almost any number of plaintext units can be permuted with minimum computing power.

A plaintext unit may be a byte, a 16 or 32 bit word or a larger ordered collection of bits. One permutation table may contain e.g. $n$=16 entries. For this example a maximum of $n! = 16! = 20922789888000$ different tables with 16 entries exist. Only a subset of several hundred or thousand such tables can be created and subsequently stored in RAM. The data structure that helps selecting tables and that adds an offset to each table is a binary tree as shown below.

The example shows a tree that has a maximum permutation capacity of 128 plaintext units (words). If each plaintext unit is e.g. 64 bit in size, a total of $128 * 64 = 8192$ bit can be permuted with only eight 16-word tables and a tree depth of $log_2 8 = 3$. Execution time of the entire word permutation operation is $O(n \cdot log_2 n)$. During key setup, 0-word or 16-word offsets (shown in violet color) and pointers to tables are assigned to the bottom tree nodes. Any of the 8 available permutation tables, each containing a valid permutation sequence to permute 16 words, can be assigned to any available bottom node. The “father” nodes of the bottom tree nodes are assigned 0-word or 32-word offsets and the respective “father” node(s) are assigned the double of 0-word or 32-word offsets during key setup: 0-word or 64-word offsets.

In order to permute 128 words, all that needs to be done is to follow the tree for each $i; 0 \leq i < 128$.

If block length is less than a power of two, e.g. 99 words, then the binary tree is only followed in a way that the resulting offset is less than the block length. The final branch in our example is: $+64 + 32 + 0 = 96$ . The “final” table, table 1, cannot be used as the mapping would be inconsistent in the end. A smaller and consistent permutation table is used in this case.

A binary tree is thus a powerful tool to make a small permutation mapping practical for universal computers as target machines. Universal computers are typically equipped with gigabytes of RAM and stack memory, which is very practical for handling binary trees.

Ternary or quaternary or any other higher-order trees may be used in lieu of a binary tree. As today’s computers operate best with powers of two, any number of “child” nodes that can be divided by two without remainder is suitable.

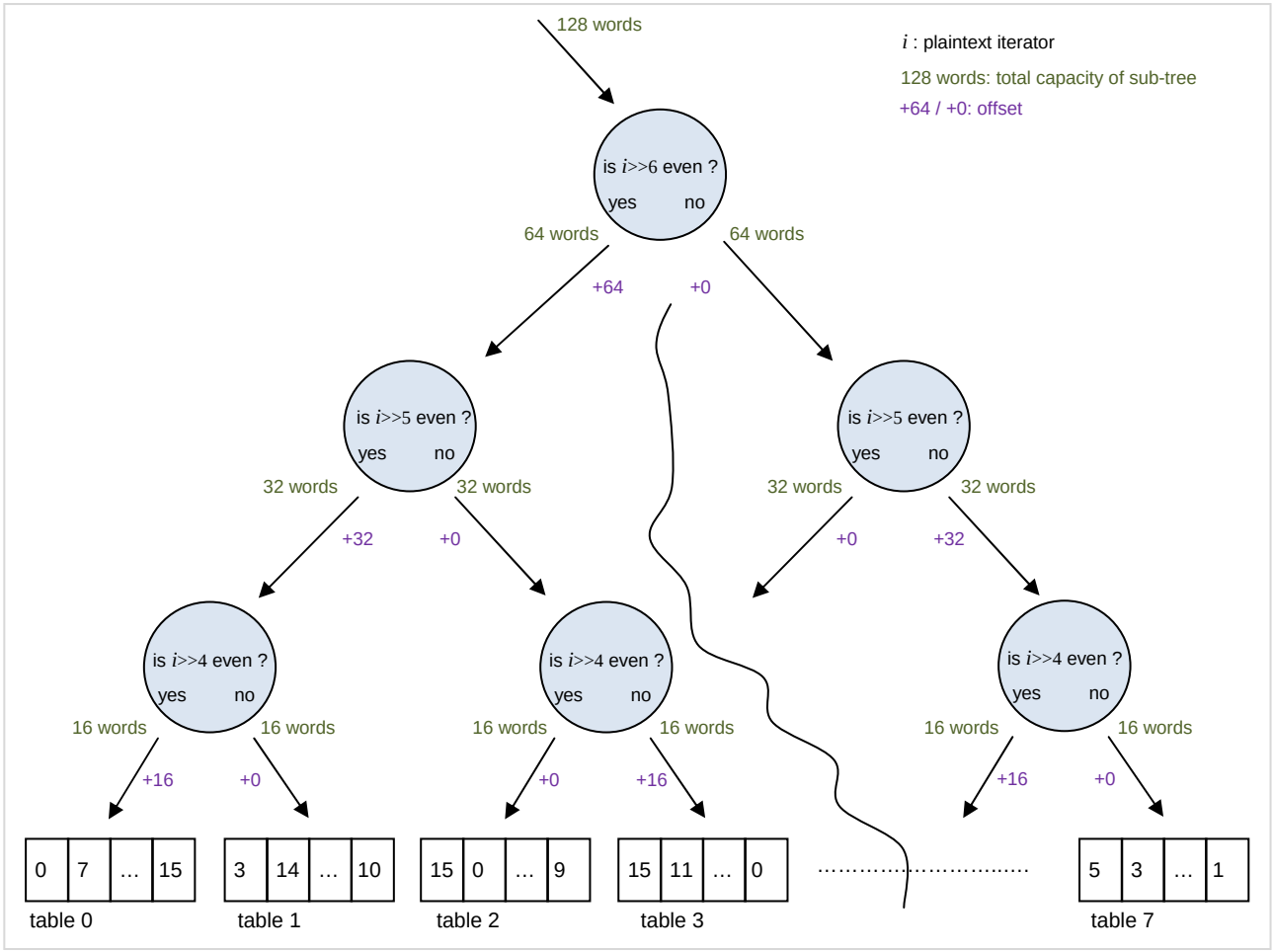


*Figure 7: Binary permutation tree*

### 2.3 Partitioning of plaintext into blocks of different size in a keyed operation

In order to deprive opponents of the knowledge about block boundaries, big, as well as small blocks can be partitioned into slices of different size. As this operation solely depends on the key and the size of the plaintext block, a good implementation of the generator that determines block sizes, forces opponents who attempt to attack the cipher by brute force to perform their decryption attempts with lots of different, but equally probable block sizes. Opponents using dedicated code-breaking hardware (e.g. an array of wafer-scale code breakers) need to make available all the resources required for blocks with maximum possible size, which can cost enormous amounts of valuable chip space for a hardware code breaker.
Computation of block sizes can be made extremely computationally expensive, e.g. by employing modular exponentiation during key setup:
Let $L_K$ and $R_K$ denote the left part and right part of a $(n_K+m_K)$ – bit string representing key $K = L_K \bullet R_K$ . We further choose a prime number $p$ in another keyed operation and compute the key $K_{BS}$ for the keyed operation that computes block sizes:

$$K_{BS} = {L_K}^{R_K} mod\ p$$

An opponent needs to perform this operation every time he tries a new key. The legitimate user of the cipher although will perform the operation only once.
Computation of the key can be rendered much more complicated e.g. by nesting the modular exponentiation operation and/or by forming an additive and/or multiplicative secrecy system.

Compared with the128 bit block size of AES Rijndael and 64 bit for DES (Data Encryption Standard) might a huge block size look oversized by several orders of magnitude. When encrypting large files or some other large amount of data, this concept is although extremely powerful. An opponent has to deal with blocks of unknown size and, if polymorphic functions are used in an actual implementation, an unknown cipher on top of this. Blocks will be typically so huge that it's impossible to mount any kind of codebook attack. Even more

important is the fact that the size of the first piece of a truncated block is unknown. This is particularly important when using the cipher in CBC mode (Cipher Block Chaining). Opponents are deprived of the knowledge about block boundaries, which even makes exhaustive sieve (Brute Force Attack) much more costly. The first block might be 531 bytes, 1 Kbyte, 10 Kbyte, 20 Kbyte or, say, 51395 bytes long.

### 2.4 Execution of three unbalanced Luby-Rackoff rounds [5]

In order to speed up encryption/decryption of giant blocks, it is desirable to be able to partition the encryption operation. The vast majority of CPUs for PCs sold today are equipped with two, four or even more processor cores with dedicated first- and often as well with dedicated second level caches. The required overhead for synchronizing a number of threads that run on different processor cores renders parallel encryption senseless for small blocks like for AES Rijndael.

In order to assure confusion [1], diffusion [1], as well as completeness [2] and maximization of the avalanche effect [3] all at a time, three-round Luby-Rackoff [5] with plaintext-biased round functions is the ideal construction. If a cryptographic transformation is complete, then each ciphertext bit must depend on all of the plaintext bits. After only two rounds, every bit depends on every other bit in the block. The construction further comes with provable security.

All of the three rounds can be partitioned so that they can be executed on different processor cores. Speed performance is best if left and right binary strings have different sizes. While the left string of round 2 may e.g. only be 256 bit long in practice, the right string may feature a length of several thousand or hundreds of thousand bits.

Three-round Luby-Rackoff [5] is an ideal construction to apply Interpreted Polymorphic Encryption as the mathematics behind it allows for a great amount of flexibility for the one-way functions used as round functions.

#### 2.4.1 The Extended Luby-Rackoff Construction

Pseudorandom function generators cannot be directly used for block encryption because they are not invertible. Luby and Rackoff were however able to show that there is a way to do so. In 1992 Maurer [4] provided a simplified explanation, which is cited and generalized in the next paragraph.
Three-round Luby-Rackoff is a process that comprises a sufficiently high number of operations so that an interpreter for a Polymorphic Encryption Algorithm won't consume an excessive amount of CPU time on the interpretation of atomic tokens. Effective linearity according to Dunkelmann and Keller [7] is the same as of random permutations, which is an ideal design goal for block ciphers.

Luby and Rackoff [5] showed that a provably secure block cipher can be constructed from just three good pseudorandom functions that are used as round functions in a Feistel structure reduced to only three rounds. This paragraph contains a generalized version of the mathematical proof, which largely consists of a citation of [4]. The following paragraphs will be dedicated to the pseudorandom functions which are used as round functions.

Let $\{0,1\}^n$ denote the set of binary strings of length $n$, let $F^n$ denote the set of all $(2^n)^{2^n} = 2^{n2^n}$ functions $\{0,1\}^n \to \{0,1\}^n$, and let $P^n$ denote the subset of functions of $F^n$ that are permutations of $\{0,1\}^n$. For $f_1 \in F^n$ and $f_2 \in F^n$, $f_1 \circ f_2$ denotes the composition of $f_1$ and $f_2$ : $f_1 \circ f_2(x) = f_2(f_1(x))$ .

For two binary strings $a$ and $b$, $a \bullet b$ denotes their concatenation. If $a$ and $b$ have the same length, $a \oplus b$ denotes their bitwise *exclusive or* combination.

Motivated by the Feistel round structure of the DES cipher, Luby and Rackoff defined a mapping $H$: $F^n \times F^n \times F^n \to P^{2n}$ assigning every triple of functions in $F^n$ a permutation in $P^{2n}$. In other words, three functions $F^n$ working with binary strings of length $n$ are combined to create a set of permutation functions

$H$ of $P^{2n}$ that map binary strings of twice the length $n$.

In our case we define for the mapping $H$: $F^n \times F^m \times F^n \to P^{n+m}$ assigning the set of functions in $F^n$, $F^m$ a (non-invertible) hash function in $P^{n+m}$. In other words, two functions $F^n$ working with binary strings of length $n$ and one function $F^m$ working with binary strings of length $m$ with $n < m$ are combined to create a set of hash functions $H$ of $P^{n+m}$ that map binary strings of length $n+m$. Hash functions have the ability to compress information, but also to expand information. This is exactly required for the general case of an asymmetric Luby-Rackoff construction.

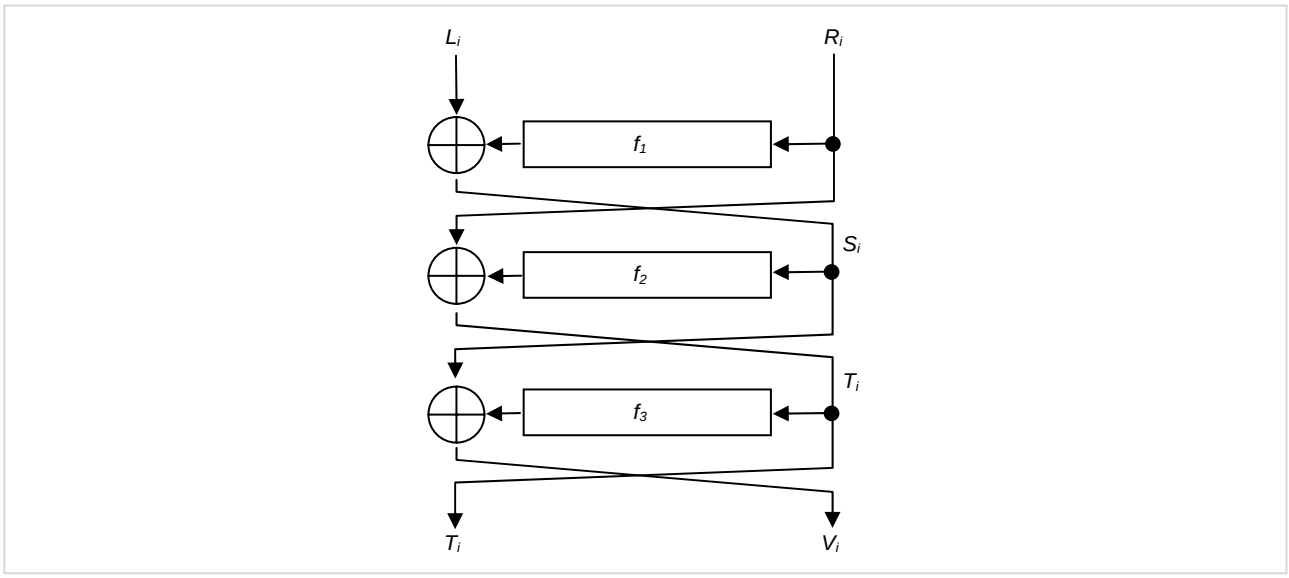


*Figure 8: Three-round Luby-Rackoff construction*

Mathematically, this mapping looks as follows:

Let $L$ and $R$ denote the left part and right part of a $(n+m)$ – bit string $L \bullet R$ and let for $f \in F^{n+m}$ the permutation $\overline{f} \in P^{n+m}$ be defined as

$$\overline{f}(L \bullet R) = R \bullet \left[L \oplus f(R)\right],$$

i.e., the right part of the argument appears unchanged while the left part of the result equals $L \oplus f(R)$. This corresponds in principle with one round of DES (Data Encryption Standard), with the difference that left and right side are unbalanced and that binary strings $n$ and $m$ with variable length are being processed.

For a list of functions, $f_1, f_3, ..., f_s \in F^n$ and $f_2, f_4, ..., f_{s-1} \in F^m$, let the permutation function $\psi(f_1, ..., f_s) : \{0,1\}^{n+m} \to \{0,1\}^{n+m}$ be defined by

$$\psi(f_1, f_2, ..., f_{s-1}) = \bar{f}_1 \circ \bar{f}_2 ... \circ \bar{f}_{s-1},$$

i.e., $\psi(f_1, ..., f_s)(L \bullet R) = \bar{f}_s\left(\bar{f}_{s-1}\left(...\bar{f}_1(L \bullet R)...\right)\right)$. The mapping H can now be exactly defined by $H(f_1, f_2, f_3)(L \bullet R) = \psi(f_1, f_2, f_3)$ (cf. Figure 2), where

$$\psi(f_1, f_2, f_3)(L \bullet R) = \left[R \oplus f_2\left(L \oplus f_1(R)\right)\right] \bullet \left[L \oplus f_1(R) \oplus f_3\left(R \oplus f_2\left(L \oplus f_1(R)\right)\right)\right].$$

The decisive question is the security of this construction. Luby and Rackoff broke this problem down to calculating the probabilty for being able to distinguish $F^{2n}$, which is $F^{n+m}$ in our general case, from a function randomly chosen from the much smaller set $\psi(F^n, F^m, F^n)$. An oracle circuit is supposed to do

this job. An oracle circuit $C_{n+m}$ is a circuit with gates consisting of $n+m$ input and $n+m$ output gates where all oracle gates in a circuit evaluate the same fixed function in $F^{n+m}$.

Let $g:(\{0,1\}^{n+m})^k \to \{0,1\}$ be a function taking as input $k$ $n+m$ bit strings. For a given set of $k$ arguments $x_1, x_2, ..., x_k$, let

$$P\left[g(f(x_1), f(x_2), ..., f(x_k)) = 1 : f \in_R \psi(F^n, F^m, F^n)\right]$$

and

$$P_g \stackrel{\Delta}{=} \left[g(f(x_1), f(x_2), ..., f(x_k)) = 1 : f \in_R F^{n+m}\right]$$

be defined as the probabilities that $g(f(x_1), f(x_2), ..., f(x_k)) = 1$ when $f$ is chosen randomly from $\psi(F^n, F^m, F^n)$ and from $F^{n+m}$, respectively.

If the two probabilities are equally likely, their difference is zero. If one of the two probabilities is less or more likely than the other, i.e. if the oracle is able to create a link between $f \in \psi(F^n, F^m, F^n)$ and $f \in F^{n+m}$, the absolute value of the difference of both probabilities is high. If an upper limit for this difference of probabilities exists and this limit is very small, three-round Luby Rackoff would be proven secure.

**Lemma 1.** *For every function* $g:(\{0,1\}^{n+m})^k \to \{0,1\}$ *and for every set of k arguments* $x_1, ..., x_k$ ,

$$\left|P\left[g(f(x_1), f(x_2), ..., f(x_k)) = 1 : f \in_R \psi(F^n, F^m, F^n)\right] - P_g\right| \le \frac{k^2}{2} \cdot (2^{-n} + 2^{-m}) .$$

*Proof of Lemma 1.* Let $f_1, f_2$ and $f_3$ be functions randomly chosen from $F^n$, $F^m$, and let $f = \psi(f_1, f_2, f_3)$. Let $x_i = L_i \bullet R_i$ for $1 \le i \le k$ be the $k$ arguments of $f$, and define $S_i, T_i$ and $V_i$ for $1 \le i \le k$ as follows (cf. Figure 2):

$$S_i = L_i \oplus f_1(R_i)$$

$$T_i = f_2(S_i) \oplus R_i$$

and

$$V_i = f_3(T_i) \oplus S_i .$$

Note that when the evaluation of $f$ for the argument $x_i$ is viewed as a three-round process (similar to three rounds of DES), the outputs of the first, second and third round are $R_i \bullet S_i, S_i \bullet T_i$ and $T_i \bullet V_i = f(L_i \bullet R_i)$, respectively. We may for the rest of the proof assume, without loss of generality, that the $x_i$, $1 \le i \le k$, are distinct. Choosing identical arguments provides no new information and can thus certainly not help.

Let $\varepsilon_S$ and $\varepsilon_T$ denote the events that $S_1, ..., S_k$ as well as $T_1, ..., T_k$ are distinct. Let $\varepsilon$ further be the event that both $\varepsilon_S$ and $\varepsilon_T$ occur. As a matter of consequence, $T_1 = f_2(S_1) \oplus R_1, T_2 = f_2(S_2) \oplus R_2, ...,$ $T_k = f_2(S_k) \oplus R_k$ are completely random because $f_2$ is a random function and hence $f_2(S_1), f_2(S_2), ..., f_2(S_k)$ are completely random. Similarly, if $\varepsilon_T$ occurs, then $V_1 = S_1 \oplus f_3(T_1), ...,$ $V_k = S_k \oplus f_3(T_k)$ are completely random because $f_3$ is a random function. Thus if both $\varepsilon_S$ and $\varepsilon_T$ occur, $f(x_1) = T_1 \bullet V_1, ..., f(x_k) = T_k \bullet V_k$ are completely random and thus $f = \psi(f_1, f_2, f_3)$ behaves precisely like a function chosen randomly from $F^{n+m}$.

Therefore the distinguishing probability is upper bounded by

$$\left|P\left[g\left(f(x_1), f(x_2), ..., f(x_k)\right)=1 : f \in_R \psi(F^n, F^m, F^n)\right]-P_g\right| \le 1-P[\varepsilon] .$$

We now derive an upper bound for $1-P[\varepsilon]=P[\bar{\varepsilon}]$, where $\bar{\varepsilon}$ denotes the complementary event of $\varepsilon$. $\varepsilon$ is the union of the $\binom{k}{2}$ events $\{S_i = S_j\}$ for $1 \le i < j \le k$ and the $\binom{k}{2}$ events $\{T_i = T_j\}$ for $1 \le i < j \le k$. The probability of the union of several events is upper bounded by the sum of the probabilities, and hence

$$1-P[\varepsilon]=P[\bar{\varepsilon}] \le \sum_{1 \le i < j \le k} P[S_i = S_j] + \sum_{1 \le i < j \le k} P[T_i = T_j] .$$

Since $f_1$ is a random function, and for $i \neq j$ we have

$$P[S_i = S_j] = \begin{cases} 2^{-n} & if \quad R_i \neq R_j \\ 0 & if \quad R_i = R_j \end{cases}$$

which further simplifies to yield

$$P[S_i = S_j] \le 2^{-n}$$

for $i \neq j$ simply because $S_i \neq S_j$ when $R_i = R_j$ because $f_1$ is a random function.
By a similar argument we obtain

$$P[T_i = T_j] \le 2^{-m}$$

for $i \neq j$ .

For the upper bound $1-P[\varepsilon]=P[\bar{\varepsilon}]$ we finally yield

$$1-P[\varepsilon]=P[\bar{\varepsilon}] \le \binom{k}{2} \cdot 2^{-n} + \binom{k}{2} \cdot 2^{-m} = \left(\frac{k!}{2!(k-2)!}\right) \cdot (2^{-n} + 2^{-m}) = \frac{k(k-1)}{2} \cdot (2^{-n} + 2^{-m})$$

$$\Leftrightarrow \quad 1-P[\varepsilon] \le \frac{k(k-1)}{2} \cdot (2^{-n} + 2^{-m})$$

As $k(k-1) < k^2$ , Lemma 1 follows.

It should be noted that the symmetric case with $n=m$ features optimum attack security. In case with $n=m$ the term simplifies to yield

$$\Leftrightarrow \quad 1-P[\varepsilon] \le \frac{k(k-1)}{2} \cdot 2 \cdot 2^{-n} = \frac{k(k-1)}{2^n}$$

Security depends mainly on $n$ as this number of bits is smaller than $m$. If the left side is only $n=64$ bits in length and the right side is much longer, e.g. $m=1024$ bits, an attack would certainly be mounted on the left side as an attack complexity of $2^{64}$ is very well manageable these days.
It is very important to note that it is always possible to divide left and right side into pieces with almost identical length! The actual demo C++ code has been programmed exactly this way. A 1397 byte block is thus divided into a 704 byte left side and a right side with 693 bytes. Attack security corresponds with the one of a 1386 byte (= $2 \cdot 693$ byte) symmetric three-round Luby-Rackoff cipher and is consequently only a little weaker than what would be theoretically possible to yield.

As long as the functions $f_1, f_2$ and $f_3$ are pseudorandom functions, the three-round Luby-Rackoff construction features proven security. The proof of Lemma 1 works just as well for permutations (that are invertible) as for pseudorandom functions that are not invertible. Good stream ciphers, but also hash functions and fast block ciphers used in Output Feedback Mode are all suitable and have been successfully used in the past to create good Luby-Rackoff block ciphers.

### 2.5 Execution of the first and the third Luby-Rackoff rounds with a short left binary string and a long right binary string

In this paper an asymmetric (unbalanced) Luby-Rackoff construction is proposed. It is taken into account that this construction is only roughly as secure as a Luby-Rackoff construction with two times the binary string length of the shorter binary string used in the configuration that is proposed here. The advantage of using asymmetric string lengths lies in the ability to execute rounds 1 and 3 fast by partitioning these computationally expensive rounds into a number of independent operations. As the substring $R_i$ is very often longer than substring $L_i$ , round functions $f_{11}$, $f_{12}$, $f_{13}$, as well as $f_{31}$, $f_{32}$ and $f_{33}$ must process longer binary strings than round function $f_2$ before combination with the first data bit using bitwise *exclusive or* or *add/subtract* or a more complex function can take place.

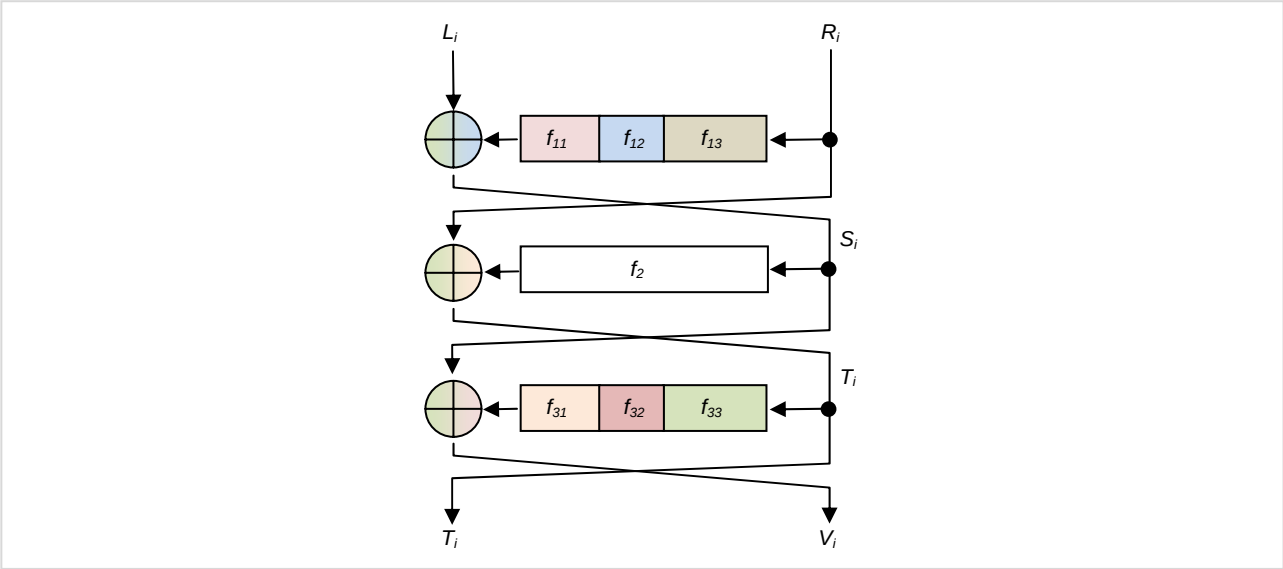


*Figure 9: Three-round Luby-Rackoff construction with partitioned rounds 1 and 3*

The figure below shows that combination, as well as computation of the round functions for round 1 and 3 take place independent of each other. Round functions $f_{11}$, $f_{12}$, $f_{13}$, as well as $f_{31}$, $f_{32}$ and $f_{33}$ all use the entire string $R_i$ ($T_i$ respectively) for their computation.

For the demo C++ code, partitioning has not been realized. The 64 bit code although still outperforms 64 bit compiled AES by approx. 50%.

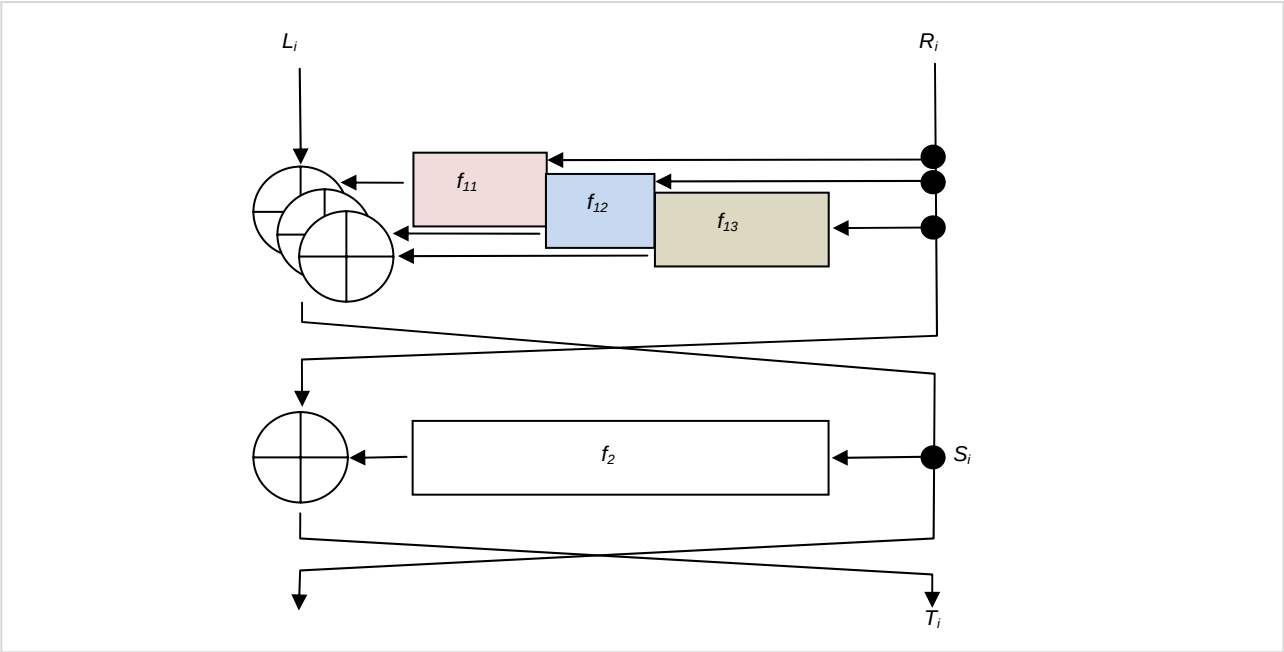


*Figure 10: Partitioned Luby-Rackoff round 1 to enable for parallel computation*

It is important to reflect that for all functions $\overline{f}(L \bullet R) = R \bullet \left[L \oplus f(R)\right]$ , the bitwise *exclusive or* combination in

$$S_i = L_i \oplus f_1(R_i)$$

$$T_i = f_2(S_i) \oplus R_i$$

and

$$V_i = f_3(T_i) \oplus S_i \,.$$

can theoretically be divided into as many bitwise operations as the respective string length. It definitely makes sense to take advantage of the longest word length that is available on the target machine, which is currently 64 bits for PCs and 32 bit for high-end microcontrollers.

Useful round functions are stream ciphers and block ciphers employed in a Davies-Meyer-, Matyas-Meyer-Oseas- or Miyaguchi-Preneel compression function (hash function) or a block cipher can be used in output feedback mode. Fast key setup and fast generation of the confusion sequence are a plus.
Round functions are discussed in chapter 3.

### 2.6 Execution of a second Luby-Rackoff round with a long left binary string and a short right binary string

Useful round functions are again stream ciphers, block ciphers employed in a Davies-Meyer-, Matyas-Meyer-Oseas- or Miyaguchi-Preneel compression function (hash function) or a block cipher used in output feedback mode. As the bias $S_i$ for round function $f_2$ is quite short, key setup is easy to perform rapidly. Fast generation of the confusion sequence is even more important for the actual implementation of a round function $f_2$ .
Round functions are discussed in chapter 3.

The entire partitioning concept can be implemented as well with long left binary strings for rounds 1 and 3 and short left binary strings for round 2 respectively. It all depends on the overall speed at which the entire construction can be executed and it may be clever to select this mode of operation dynamically by making the selection a keyed operation.

### 2.7 Encryption of excess bytes

In order to render the cipher as efficient as possible, 64 bit operations are preferably used. As a matter of consequence, the minimum size of an ordered collection of bits is 64 for a 64 bit implementation. In order to encrypt 1 .. 7 excess bytes, special treatment of this data is required:
The best solution is simply to combine excess bytes in round 2. By doing this, even the Strict Avalanche Criterion is met.

### 2.8 Encryption of short blocks

Blocks that are too short for encryption using partitioned Luby-Rackoff (less than e.g. 256 bit) require special treatment. Although system integrators are strongly discouraged to encrypt single bytes, 64 bit blocks or 128 bit blocks, the proposed cipher may contain functionality to encrypt such small amounts of data. Blocks sized between 128 .. 512 bit are encrypted by a shrinked version of the giant block size cipher proposed in this paper. As a whole host of attacks are applicable to ciphers with small S-boxes, it may be wise not to implement support for encrypting small blocks. As an example, when encrypting more than 256 8 bit blocks with the very same key, an adversary might already have built up a codebook. The only precaution against codebook attacks is to force the adversary to build up an incredibly big and expensive codebook that he might never be able to fill with data. 128 bit is regarded as being safe today.

### 2.9 Alternative bitwise combination functions

Bitwise *exclusive or* is the simplest operation suitable for combining data with a confusion sequence. The logical operation is invertible. The same confusion sequence and the same logical operation is used for encryption as well as for decryption.
It is although as well possible to take advantage of *exclusive or* and *add/subtract* chosen in a keyed operation. An *add* operation used to encrypt a word is inverted by using a *subtract* operation during decryption and vice versa. *Exclusive or*, *add*, as well as *subtract* execute in a single machine instruction on probably all modern microprocessors.
With only one additional machine instruction, 128 equally likely permutations of 64 bit words can be added through the use of bitwise left or right rotation (which is NOT to be mistaken for arithmetical shift!). This yields $3 \cdot 128 = 384$ equally likely combination functions for the combination of 64 bits at a time.

## 3. Round functions: Generation of High-Quality Pseudorandom Bitstreams

The decorrelation module as well as the round functions in a Luby-Rackoff structure require good pseudorandom functions, preferably ones that are impossible to analyse. This is where the strength of classic polymorphic pseudorandom number generation is found. A polymorphic pseudorandom number generator executes conceptually different pseudorandom number generators (PRNG) in an order that is defined by the key and by the shared internal state. By initializing the shared internal state with the entire key and an initialization vector and by defining the sequence of operations with the key, one can easily parameterize this multiplicative/additive combined secrecy system.
The basis of a polymorphic pseudorandom number generator is typically a set of lightweight PRNGs. It is even possible to use cryptographically weak functions like LCG, which is still in use in many systems as source for pseudorandom numbers.
It is possible to create polymorphic PRNG functions that are initialized within a few clock cycles and that generate very fast streams of pseudorandom numbers - as required for round function $f_2$. For the other round functions, polymorphic PRNG functions are required that can be initialized as quickly as possible with a large and variable bit stream and that output a rather small number of pseudorandom numbers.

### 3.1 Dynamically selected Pseudorandom Number Generators

Unknown but cryptographically weak pseudorandom generators can only be identified by looking at their output sequence. A real spectacular example is the Linear Congruential Generator $x_{i+1} \equiv (3x+5) \bmod 15$ . For the start value $x \equiv 7$ , the output sequence becomes $7,11,8,14,2,11,8,14,2,11,...$ . Without knowing the

recurrence relation $x_{i+1} \equiv (3x+5) \bmod 15$, it is possible to identify this relation after taking only a few samples.

Let $P_b[Y_{i+b} = y_{i+k}]$ be the probability that an oracle can guess output values $Y_i$ after recording and analysing the last $b$ output values; after $k$ output values have been recorded, let the oracle be able to identify the pseudorandom number generator $RNG$ and to predict all following output values.

For $P_b[Y_{i+b} = y_{i+k}]$ we have:

$$P_b[Y_{i+b} = y_{i+k}] == \begin{cases} 1 & if \quad b \geq k \\ 0 & if \quad b < k \end{cases}$$

**Theorem 1.** *Let $Z$ and $Y$ denote left and right parts of a binary string and let $Z_{i+1} \bullet Y_{i+1} = PRNG_{Yi}(Z_i \bullet Y_i)$ be the recurrence relation of a set of similar but not identical pseudorandom number generators. Before executing a pseudorandom number generator, the actual function is selected by the $Y$ part of the binary string that was output upon the last execution of a (probably different) pseudorandom number generator function. As long as a different pseudorandom number generator is selected before the oracle is able to gain sufficient knowledge about its identity, $P_b = 0$ .*

*Proof.* $P_b[Y_{i+b} = y_{i+k}]$ is defined to be 0 if $b<k$ . The oracle is given less than $k$ samples of the output sequence of one specific pseudorandom number generator function. Thus it is unable to make any prediction at all.

Another, much more hypothetical assumption for an oracle, may be the ability to predict on average every *n*-th output value of a pseudorandom number generator function, as well as the ability to predict which pseudorandom number generator function is used on average for every *m*- th output value.

Again we have for the recurrence relationship

$$Z_{i+1} \bullet Y_{i+1} = PRNG_{Yi}(Z_i \bullet Y_i)$$
$$Z_{i+2} \bullet Y_{i+2} = PRNG_{Yi+1}(Z_{i+1} \bullet Y_{i+1})$$
$$\dots$$
$$Z_{i+j+1} \bullet Y_{i+j+1} = PRNG_{Yi+j}(Z_{i+j} \bullet Y_{i+j})$$

For the probability of the oracle to be able and guess the first output value $P_b\lfloor Z_{i+1} \bullet Y_{i+1} = y_g \rfloor$ , and by assuming that the value $Z_i \bullet Y$ is known, we yield

$$P_b[Z_{i+1} \bullet Y_{i+1} = y_g] = \frac{1}{n}$$

which is identical to the probability to guess each of the parts of the binary output string

$$P_b[Z_{i+1} \bullet Y_{i+1} = y_g] = P_b[Z_{i+1} = y_g] = P_b[Y_{i+1} = y_g] = \frac{1}{n}$$

Due to the fact that the oracle needs to predict the actual function as well as the output value, we yield for the prediction probability of the oracle after executing the recurrence relationship two times

$$P_b[Z_{i+2} \bullet Y_{i+2} = y_g] = \frac{1}{n} \cdot \frac{1}{m}$$

After $j$ executions we yield

$$P_b[Z_{i+j+1} \bullet Y_{i+j+1} = y_g] = \frac{1}{n} \cdot \left(\frac{1}{n} \cdot \frac{1}{m}\right)^{j-1}$$

Even if an oracle as powerful as outlined above was available to an attacker, the sequential execution of a Polymorphic Pseudorandom Number Generator can potentially render comparably insecure base functions into powerful sources of pseudorandom numbers. It should be noted that if the very same pseudorandom number generator function $PRNG_{Yi}$ was used again and again, the probability of the oracle to be able and guess output values would remain at $1/n$ .
A clever design of a pseudorandom number generator using dynamic base function selection will not only take history into account for the selection of base functions, but also keying information and, if available, other data.

### 3.2 Compiled Pseudorandom Number Generator stack forming a multiplicative/additive combined secrecy system

Very fast Polymorphic Cipher designs so far have always relied on the strength of compiled cryptographic base functions. A crypto compiler is used to compile an algorithm directly from a key. Each key thus generates one unique cipher or a stack consisting of different pseudorandom number generators (PRNGs). Throughout the first part of this chapter it is assumed that a crypto compiler compiles identical Linear Congruential Generator (LCG) primitives to form a PRNG stack that operates with an internal state that is shared by all compiled PRNGs. Although LCG PRNGs are very rarely implemented in actual Polymorphic Ciphers, this weak PRNG is perfectly suited to show plainly the strength of multiplicative/additive PRNGs.

The stacked LCG PRNGs are supposed to pass information from one primitive to the next in the stack.

The linear congruential sequence of a pure multiplicative LCG is determined by ($a$, $X_n$ and $M$).

$$X_{n+1} = (a \cdot X_n) \bmod M$$
$$X_{n+2} = (a \cdot X_{n+1}) \bmod M$$
$$X_{n+3} = (a \cdot X_{n+2}) \bmod M$$
$$\ldots$$

As there are three unknowns ($a$, $X_n$ and $M$), consequently three consecutive samples are sufficient to break this generator.

If used with a randomiser, the task to break a modified LCG primitive in a compiled PRNG can be described as

$$X_{n+1} = (r(n) \cdot X_n) \bmod M \;\; ;$$

with r(n) being a sequence of numbers
randomly selected by the crypto compiler

yielding the congruential sequence

$$X_n = (r(0) \cdot X_0 + r(1) \cdot X_1 + \ldots + r(n) \cdot X_n) \bmod M$$

The minimum number of samples required to determine all unknowns equals $n+2$ . The unknows are $r(0)$, $r(1)$, .., $r(n)$, $X_0$, $M$. An opponent gets $n$ samples to try and break the stack but has to deal with $n+2$ unknowns. His task cannot end successfully.

LCGs are good examples for base functions that are comparably insecure, but that can be hardened by using them in a stack of compiled base functions. Almost any function can be added to such a stack – even complete ciphers like DES, Magenta, RC6 or AES. Such base functions are very easy to parameterize: The crypto compiler simply assigns a key to such base functions.

Faster and much smaller base functions that can be stacked more often than slow and complex base functions include:

Add-with-carry generators (ACG):

$$X_n = (X_{n-s} + X_{n-r} + \text{carry}) \bmod M$$

These generators have long periods, easily exceeding $10^{200}$, and they are even faster than LCGs.

Multiply-with-carry generators (MWCG) use this simple relation:

$$X_n = (\text{a}X_{n-1} + \text{carry}) \bmod M$$

Multiplier $a$ can be chosen from a large set of integers without affecting the period of around $2^{31}$-1 for 32 bit implementations. MWCGs easily pass standard randomness tests.

Add-with-carry generators can feature a very long period if $s$ and $r$ are large:

$$X_n = (X_{n-s} + X_{n-r} + \text{carry}) \bmod M$$

If a sufficiently large number of such primitive PRNGs are concatenated to form one single PRNG, security holes of each primitive PRNG are levelled out. Such a combined secrecy system has the unique feature to exhibit no static weakness and it overwhelms an opponent with a large number of variables. The number of variables is at any time greater than the number of knowns.

A limited number of PRNGs form a polymorphic pseudorandom function $f(x)$ with the following program-controlled recurrence relation:

$$X_n = \begin{cases} LCG(X_{n-1}, a) & if \quad PROG\_SEQ[n] = 0 \\ ACG(X_{n-1}, s, r) & if \quad PROG\_SEQ[n] = 1 \\ \ldots & if \quad \ldots \\ MWCG(X_{n-1}, a, carry) & if \quad PROG\_SEQ[n] = m \end{cases}$$

with *a* ,*s* ,*r* and *carry* being numbers selected pseudo-randomly by the crypto compiler and *PROG_SEQ[ ]* representing a program sequence. There shall exist m conceptually different PRNG base functions

The program sequence $PROG\_SEQ[\ ]$ can be of almost arbitrary size. Initialization of this sequence is a keyed operation that can be performed during setup of the encryption context. The process is preferably part of the key expansion step.

The polymorphic pseudorandom function $f(x)$ is the sum of $m$ PRNGs. This system can be described as the sum of operations $O$ with each $O_i$ being a specific PRNG corresponding to key choice $i$, which has probability $p_i$:

$$O = p_1 O_1 + p_1 O_1 \ldots + p_m O_m$$

When executing the polymorphic pseudorandom function $f(x)$ repeatedly, the multiplicative system $P$ is formed:

$$P(n) = O(n)O(n-1)\ldots O(0)$$

It should be noted that $P$ is not commutative. After $n$ iterations, one out of $nm$ different multiplicative/additive PRNG systems has been executed.

The principle can additionally be employed to create a polymorphic pseudorandom function $f(x)$ that resists power attacks (SPA, DPA). A battery of almost identical PRNGs that compile into almost identical machine code is employed. The following assumptions need to be made for almost identical machine code:

- Execution times of interchanged instructions must be identical, which is the case for *add, subtract* and *exclusive or* operations
- Power consumption of interchanged instructions must be identical, which is again the case for *add, subtract* and *exclusive or* operations

### 3.3 Fast Polymorphic PRNG Functions for the generation of large pseudorandom bitstreams with short bias bitstreams

For a round function $f_x$ it may be important that the internal state can be copied quickly from the initial internal state and that it runs at high speed.

One possible configuration is a polymorphic PRNG function with a big key-dependent internal state, a long polymorphic sequence and an additional "Immediate" internal state that is only a few words in size and that is used by every polymorphic base function to compute pseudorandom numbers.

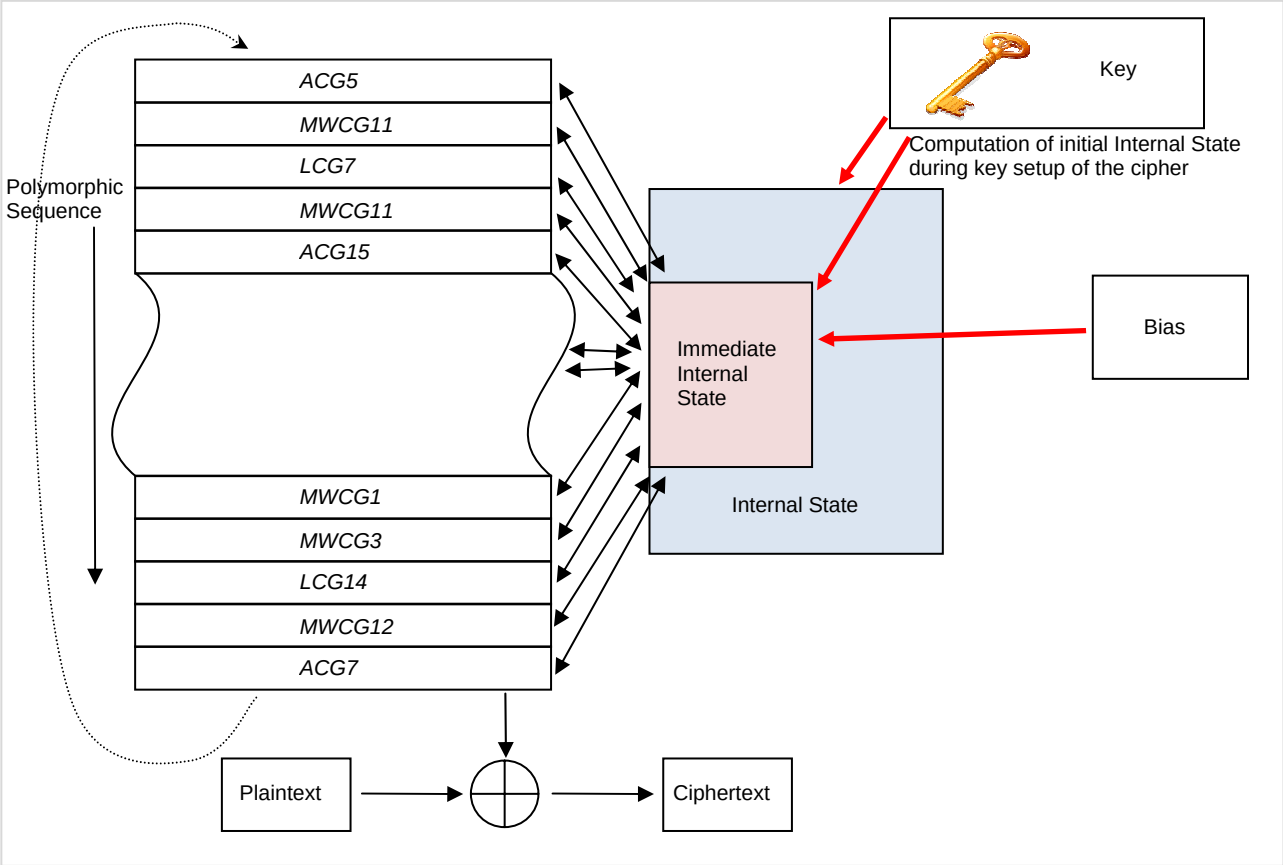


*Figure 11: Polymorphic PRNG function with fast bias key setup for round function $f_1$, $f_2$ or $f_3$*

The proposed polymorphic PRNG executes a sequence of base PRNG functions like ACGs and MWCGs that all share the same internal state where all parameters for the base PRNG functions are stored. This internal state constantly changes in a pseudorandom way during execution of the sequence of base PRNGs. Every base PRNG function requires a number of parameters and history in order to be suitable. A plain LCG PRNG does not satisfy this requirement. With a few modifications, i.e. an additional operation that at least modifies the Immediate Internal State, even this totally insecure base function would be suitable.

During key setup with a potentially long key, the sequence of operations is determined and the entire Internal State is initialized. This first key setup is executed during key setup of the entire cipher.

As soon as e.g. round function $f_2$ is invoked, the Immediate Internal State is bitwise *exclusive ored* with $S_i$ (which acts as bias). This second key setup step can be executed within only a few machine instructions. After the second key setup step is the proposed polymorphic PRNG immediately able to generate high-quality pseudorandom numbers (if properly designed).
Prior to the second key setup step it is necessary to reset the Immediate Internal State.

### 3.4 Fast Polymorphic Hash Functions for the generation of short pseudorandom bitstreams from long bias bitstreams

For the round functions $f_{1x}$ and $f_{3x}$ , strings $R_i$ and $T_i$ need to be compressed. The result of this operation is then bitwise *exclusive ored* with $L_i$ and $S_i$ .
The requirements for these round functions differ greatly from those for round function $f_2$ .

In this case the operating principle of a hash function satisfies the requirements in the best way.

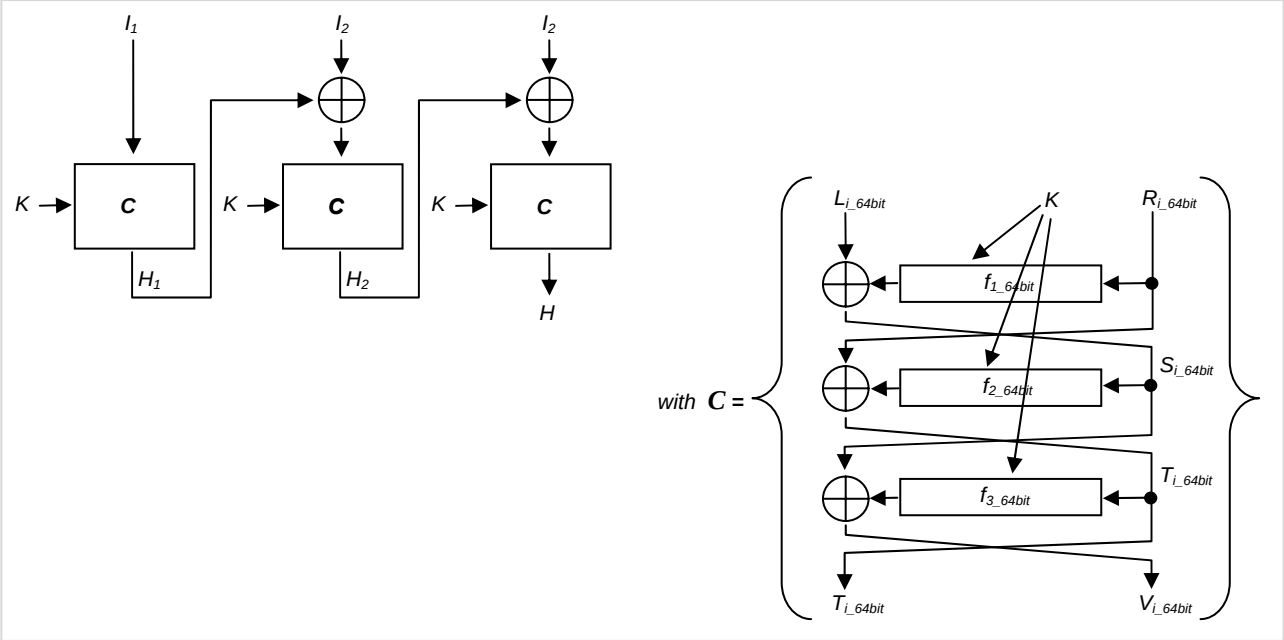


*Figure 12: Hash function based on a three-round Luby-Rackoff construction in a CBC-MAC configuration*

Although execution time of this keyed operation is $O(n)$, three entire Luby-Rackoff rounds need to be executed per 128 bit word. It is desirable to take advantage of 64 bit instructions so that cipher $C$ outputs 128 bits at a time. With three round functions $f_{11}$, $f_{12}$, $f_{13}$, as well as $f_{31}$, $f_{32}$ and $f_{33}$ running in parallel in different threads, 384 signature bits can be computed in parallel at a time. Provisions need although to be taken that the round functions $f_{11}$, $f_{12}$, $f_{13}$ (and $f_{31}$, $f_{32}$, $f_{33}$ respectively) can as well be executed in the context of one or two threads only. It only makes sense to compute round functions in parallel if processing time is saved. As synchronization time is inevitably spent, parallel execution only makes sense if there's a benefit. This is the case for big blocks that are encrypted in a single piece.

## 4. Conclusion

The proposed Polymorphic Giant Block Cipher is a groundbreaking solution for the encryption of typical IP data packets as well as files. While popular block ciphers that are heavily promoted by Secret Services feature tiny block sizes compared with the size of today's user data packets and thus don't exhibit much avalanche, does the proposed new class of Polymorphic Encryption Algorithms break the block size confinement of conventional block ciphers while satisfying the Strict Avalanche Criterion in an unprecedented way for a wide range of block sizes without changing the original plaintext size. This new method is perfectly suited to encrypt blocks with variable sizes like TCP/UDP data packets. There is no need

to pad remaining plaintext bytes with useless information. The Polymorphic Giant Block Cipher thus solves the issues that are associated with large block sizes.

## References:


[1] C.E. Shannon. Communication theory of secrecy systems. Bell System Technical Journal, 1949

[2] H. Feistel. Cryptography and Computer Privacy. Scientific American, Vol. 228, No. 5, 15 (1973)

[3] A.F. Webster, S.E. Tavares. On The Design of S-Boxes, 1986.

[4] U. M. Maurer. A simplified and Generalized Treatment of Luby-Rackoff Pseudorandom Permutation Generators. EuroCrypt '92, Springer LNCS v.658, pp.239-255, 1992

[5] M. Luby, C. Rackoff. How to construct pseudorandom permutations from pseudorandom functions. SIAM Journal on Computing, Vol. 17, No. 2, pp 373-376, 1988

[6] M. Naor, O. Reingold, On the construction of pseudo-random permutations: Luby-Rackoff revisited, Journal of Cryptology, Vol. 12, pp. 29-66, 1999

[7] Orr Dunkelman, Nathan Keller. A New Criterion for Nonlinearity of Block Ciphers. http://vipe.technion.ac.il/~orrd/crypt/apnp.pdf, 2006

[8] Nicolas T. Courtois, Josef Pieprzyk. Cryptanalysis of Block Ciphers with Overdefined Systems of Equations http://eprint.iacr.org/2002/044.pdf, 2002

[9] A. Biryukov, D. Khovratovich, Related-key Cryptanalysis of the Full AES-192 and AES-256, https://cryptolux.org/mediawiki/uploads/1/1a/Aes-192-256.pdf, 2009

[10] S. Chari, C. Jutla, J.R. Rao, P. Rohatgi. A cautionary Note Regarding Evaluation of AES Candidates on Smart-Cards. http://citeseer.nj.nec.com/chari99cautionary.html, 1999

## For more information: http://www.pmc-ciphers.com