ॐ भूर्भुवस्व तत् सवितूर्वरेण्यम् भर्गो देवस्य धीमहि धियो यो न प्रचोदयात्

EnRUPT: First all-in-one symmetric cryptographic primitive.

Sean O'Neil
www.enrupt.com
[Last updated: 02.06.2008]

Abstract: EnRUPT is a new small and simple scalable word-based symmetric cryptographic primitive that can be used to construct fast and secure block ciphers, hash functions, stream ciphers, RNGs and MACs. It accepts keys and data blocks of any size. It is currently the simplest of all the block and stream ciphers and hash functions. It is very easy to memorise. It was found and tested using automated cryptanalysis. It does not use any patented or patentable techniques. It is a nearly optimal word-based "ADD-XOR-ROL" design forming an unbalanced Feistel network first proposed in XXTEA. EnRUPT design, all its implementations and pictures are in the public domain.

1. Introduction

Block ciphers (TEA, XTEA, XXTEA, RC5), stream ciphers (Salsa20, Phelix) and hash functions (MDx, SHA, RipeMD) based on ADD-XOR-ROL networks have been studied for years. Despite their incredible complexity, many of them were found insecure. It calls for new simpler, faster and more secure primitives.

EnRUPT is a scalable unbalanced Feistel network first proposed in XXTEA[1]. It operates on w-bit words in little-endian format. The recommended word width w is currently fixed at 32 or 64 bits. EnRUPT has four parameters: 1) source block k (the key), 2) its size kw, 3) target block x (the state), and 4) its size xw. Security of EnRUPT is measured by the smallest of the two blocks: $2^{w*\min(xw,kw)}$, but for some applications it is naturally bound by the birthday barrier or by the size of the secret. When data is to be encrypted, it must be supplied as the state. When it is to be hashed irreversibly, it must be supplied as the key and the initial hash value as the state. The reversible EnRUPT round function can be summarised as:

$$x_r \oplus= \text{rotr}\,(2*x_{r-1} \oplus x_{r+1} \oplus k_r \oplus r,\ w/4) * 9 \oplus k_r, \qquad \text{(er1)}$$

where $\oplus$ is XOR (^ in C, Java, JavaScript, Perl and Python), rotr is w-bit rotation right, * is multiplication modulo 2^w and all the indexes are modulo their specific block sizes – mod kw for the key and mod xw for the state. The (er1) round function should be iterated for $n=s*(2*xw+kw)$ rounds (normally $s=4$), with the round number r beginning with 1.

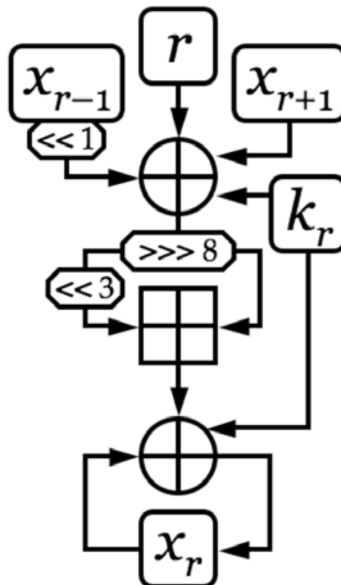

Fig 1. Block enRUPT/unRUPT round function (er1)

The key is processed irreversibly. The target block is reversible (decipherable) only when the key is known. Decryption is identical to encryption, except the state is processed iterating r in reverse. There are no special constants to memorise. The entire message can be encrypted as a single block providing free authentication of packets by verifying a constant or random value inside the block.

1.1 Notation

c	– next w-bit ciphertext value, in little-endian format
d	– previous w-bit ciphertext/keystream value, saved between rounds
k	– *key* or *source* block, in little-endian format
k_r	– *current* [r *mod* kw] word of the key
n	– number of rounds in the encryption/hashing process; $n=s*(2*xw+kw)$ for block EnRUPT and $n=s*xw$ for stream EnRUPT.
p	– next w-bit long plaintext value, in little-endian format
r	– round number, 1 to n during encryption and n to 1 during decryption
s	– security/performance trade-off parameter; $s \leq 1$ is insecure, $s=2$ defends against passive distinguishers, $s=3$ against all non-adaptive chosen plaintext, ciphertext or related key attacks and $s \geq 4$ against all adaptive attacks; $s>4*sw$ is for the high-assurance applications requiring unreasonably high security margins. We recommend $s=4$ for all modes of operation.
w	– word width, currently fixed at $w=32$ or $w=64$ bits, $w=32$ by default.
hw	– size of the hash in w-bit words, normally equal to $sw*2$ ($hb=hw*w$)
kw	– size of the source block (key) in w-bit words ($kb=kw*w$)
sw	– required security level in w-bit words
xw	– size of the target block (state) in w-bit words ($xb=xw*w$)
x	– *state* or *target* block, in little-endian format
x_r	– *current* [r *mod* xw] to be updated word of the state

2. Operation

The name must reflect all the size parameters and the required security/performance trade-off parameter s as EnRUPT*w-kb-xb-hb*/*s* to ensure clarity and compatibility with other applications. If omitted, $s=4$. If hb is omitted, $hb=kb$. If xb is omitted, $xb=kb$ in block ciphers or $2*kb$ otherwise. Names of the functions proposed in this paper are as follows:

Block Cipher	Hash Function	Stream Cipher, PRNG	MAC	Stream Hash
enRUPT/unRUPT	mdRUPT	RUPT/aeRUPT	mcRUPT	irRUPT

For the general-purpose applications, we recommend the following three types of functions with the security trade-off parameter s set to offer resistance to all adaptive attacks:

1) Block functions with keys (or data blocks in hashing) and data blocks (hashes) of the same size twice the required security: ciphers with $xw=kw=2*sw$ as in enRUPT-512 equivalent to enRUPT32-512-512/4, and MD hash functions with $kw=hw=2*sw$ and $xw=s*sw$ as in mdRUPT-512 equivalent to mdRUPT32-512-1024-512/4.

2) Functions processing whole messages as single blocks: ciphers with $kw=2*sw$ and $xw=${message words} as in enRUPT-512-max equivalent to enRUPT32-512-max/4, and MD hash functions with $xw=s*sw$, $hw=2*sw$ and $kw=${message words} as in mdRUPT-max-512 equivalent to mdRUPT32-max-1024-512/4.

3) Stream functions processing one w-bit word of plaintext/ciphertext every two rounds, with $kw=2*sw$ [$hw=2*sw$] and $xw=s*sw$, such as RUPT-512, mcRUPT-512, irRUPT-512 or aeRUPT-512, which are all equivalent to *x*RUPT32-512-1024-512/4.

Other variants may use smaller or larger states or reduced or increased numbers of rounds for increased security or increased performance. The name of course must reflect the difference to ensure compatibility with other applications. There are only two absolute restrictions: 1) state x in stream modes and block hash modes cannot be smaller than $2*sw$ words, and 2) stream modes must execute at least 2 (two) rounds between inputs/outputs.

2.1 Block cipher mode (enRUPT, unRUPT)

2.1.1) Per-block initialisation

Provide the secret key as the *source* block and its size in w-bit words as kw. If IV is required, concatenate it with the key (key first) and supply it as the *source* block.

Provide plaintext/ciphertext as the *target* block and its size in w-bit words as xw. If $xw<2$, the block cannot be decrypted. If necessary, include a constant or random authenticator value of required size as part of the plaintext target block.

Set $n = s*(2*xw+kw)$, by default $s=4$.

(if) enRUPT: Set $r = 1$.

(if) unRUPT: Set $r = n$.

2.1.2) Iteration

Execute (er1) for n rounds.

(if) enRUPT: Increment r on each round as long as $r \leq n$.

(if) unRUPT: Decrement r on each round as long as $r > 0$.

2.1.3) Finalization

(if) unRUPT: If necessary, check the authenticator value inside the decrypted block.

2.2 Stream modes (RUPT, aeRUPT, mcRUPT, irRUPT)

In stream modes, EnRUPT updates the state irreversibly as shown below. The only difference is that the key word k_r is replaced by a w-bit feedback variable d storing the previous keystream/ciphertext value linearly combined with $x_{r+\lfloor xw/2 \rfloor}$.

$t = \mathrm{rotr}(2*x_{r-1} \oplus x_{r+1} \oplus d \oplus r, w/4) * 9;$

$d \oplus= t;$

$x_r \oplus= t;$

$d \oplus= x_{r+\lfloor xw/2 \rfloor};$

$r = r+1;$

Fig 2. Stream EnRUPT round function (ir1)

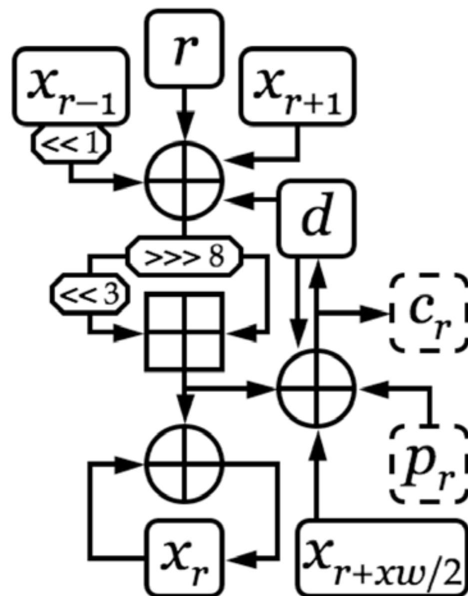

Fig 3. aeRUPT/mcRUPT/irRUPT structure

In all the stream modes, the (ir1) round function must be iterated at least twice between inputs/outputs. There is no source block, so the formula for n is also slightly different. The average period of RUPT is ~$2^{w*((xw+1)/2+1)}$, so it should be rekeyed after $2^{w*xw/4}$ outputs, although probability of such short loops is infinitesimal.

To ensure $w*sw$ bits of security, the secret state size xw must be at least $2*sw$, but we recommend $4*sw$ to provide a good security margin. The size of the final hash value hw cannot be less than $2*sw$. If it is a keyed mode of operation, the key size or concatenated key+IV size kw must also be at least $2*sw$. By default, xw=$4*sw$ and kw=hw=$2*sw$.

2.2.1) Initialisation

Provide the state size in w-bit words as xw. It cannot be less than 2.

(if) If it is a keyed function, provide the secret key or concatenation of the secret key and the IV (key first by default) as the first kw w-bit words of the input stream.

Set all xw words of the state to 0. Set $n = s*xw$, by default s=4. Set r = 1. Set d = 0.

(if) If it is a keyed function, execute (ir1) for $2*kw$ rounds incrementing r on each round to load the key [and IV] words as described in **2.2.2**. Then execute (ir1) for n more rounds incrementing r on each round.

2.2.2) Stream processing

(if) Unkeyed unrandomized irRUPT or any keyed stream mode that may have to reuse nonces with a fixed key: iterate the (ir1) round function s times.

Iterate (ir1) twice.

(if) mcRUPT/irRUPT message processing or keyed RUPT/aeRUPT/mcRUPT/irRUPT key/IV loading: set $d \oplus= p_{[r/2]}$ to load the next w-bit word in little-endian format.

(if) RUPT keystream generation or mcRUPT/mdRUPT/irRUPT output of MAC/hash: release w bits of d as keystream output.

(if) aeRUPT: release w bits of ciphertext $c_{[r/2]} = d \oplus p_{[r/2]}$ as output during encryption or w bits of plaintext $p_{[r/2]} = d \oplus c_{[r/2]}$ during decryption. Save $c_{[r/2]}$ as d.

2.2.3) Finalization (if necessary)

(if) irRUPT: pad the last word of the stream with a single bit 1 followed by zeros, or load one more word '1' if it is full, or pad the message as the protocol requires.

Execute (ir1) for n more rounds incrementing r on each round.

Execute (ir1) for $2*hw$ more rounds in keystream generation mode as described in **2.2.2** to produce the required hw w-bit words of the final MAC or hash value.

2.3 Collision-resistant block hashing (Merkle-Damgård) mode (mdRUPT)

To ensure $w*sw$ bits of security, the state size xw, the size of the final hash value hw and the size of the data block to be hashed kw must be ≥ $2*sw$ of w-bit words. If it is a keyed hash function, key or concatenated key+IV (key first by default) must also be ≥ $2*sw$ of w-bit words. By default, xw=$4*sw$ and kw=hw=$2*sw$, the same as for irRUPT.

The choice of prefix, postfix and padding is left to the protocol/standard designer. mdRUPT core only provides a secure compression/expansion function so it could replace any existing Merkle-Damgård or other constructions regardless of their high-level operation.

2.3.1) Per-message initialisation

Set all the *xw* words of the target block (hash state) to 0 [or to any other constant].

Set $n = s*(2*xw+kw)$, by default s=4. Set $r = 1$.

If it is a *keyed* hash function, execute step **2.3.3** as many times as required to hash the secret key and IV/nonce as the first source block[s].

2.3.2) Before processing the last block

If it is a *keyed* hash function, execute step **2.3.3** as many times as required to process the secret key and the IV/nonce again as the last source block[s].

If processing messages of odd sizes, pad the last word of the message with a single bit 1 followed by zeros to the end of the block, or attach one more word containing the padding '1' if the last word is full, or pad the message as the protocol requires.

2.3.3) Iteration

For each data block, execute (er1) for *n* rounds incrementing *r* on each round.

Reset *r* to 1 between blocks.

2.3.4) Finalization

Set d=0. Set r=1.

Execute (ir1) for $2*hw$ rounds in the keystream generation mode as in **2.2.2** to produce the required *hw* *w*-bit words of the final hash value.

Note: Key expansion/compression can be performed by hashing it with irRUPT, mcRUPT or mdRUPT, concatenated with other constant or random values according to the protocol.

3. Design

Formally speaking, EnRUPT is a consistent incomplete source-heavy heterogenous UFN (unbalanced Feistel network)[2]. Despite its apparent simplicity, it seems to be an optimal structure that can process equally well blocks of any size. Simplicity of every aspect of the complete design was our main goal and turned out to be the hardest thing to achieve while maintaining high performance with a single simple set of rules for all possible sizes and applications that require security for completely different attack scenarios. It is actually very easy to save a few clock cycles per byte at the cost of simplicity.

If ultra-high speed is necessary, EnRUPT can take advantage of CTR or any other parallelisable mode like any other cipher. MAC and hashing can also be parallelised 4 times as proposed in [10]. It is not the purpose of this specification to restrict the user to a certain specific mode of operation, but to provide a primitive that can be safely used in any mode.

3.1 Design principles

First of all, s-boxes, key/data-dependent rotations and multiplications were not considered at all. Key/data-dependent rotations and s-boxes suffer horribly from timing problems – fragility totally unacceptable for a cipher. Multiplications may be cheap in some modern processors, but not only they are terribly expensive in all the embedded processors and in the FPGA and ASIC hardware, they are also not nearly as fast as ADD-XOR-ROL networks at building PRFs according to our tests. This leaves only addition, subtraction, bitwise shifts, rotations and logical operations to use in the search for an optimal construction, ultimately resulting in ADD-XOR-ROL networks that are both fast and simple.

With complexity of software and hardware designs growing exponentially with time, it is important to offer developers a symmetric cryptographic instrument that they will not make a mistake implementing. Developers should not struggle to implement cryptography to secure their increasingly complex systems. Thus the cryptographic primitive should be:

1. Extremely simple
2. Easy to memorise and hard to forget
3. Almost impossible to make a mistake implementing (no timing issues either)
4. Equally strong when iterated in either direction
5. Fast on the latest most widely used software processors
6. Reasonably fast [and small] on embedded processors and in high-level languages
7. Updating x_r reversibly (by XOR) so that it could be reused as it is for decryption

3.2 Measuring polynomial pseudorandomness as design methodology

Automated algebraic distinguishers as described in [2] can measure size and randomness of polynomial relationships between all the bits. The number of rounds indistinguishable from random and the bias in the last distinguishable round can be used as the *measure of strength* of each round function. It also provides a good estimate of the number of rounds required for the cipher to provide sufficient security by combining these results with theoretical works ([4]-[9]) and with cryptanalytic results for well-studied ciphers and hash functions. Resistance to guess-and-determine attacks was measured by other tools.

Due to the sequential structure processing one word at a time, at least $xw+kw$ rounds are required even for a perfect round function of this type to propagate a change in the key or data to the entire block. Since a perfect round function would be very expensive, the second best simple enough proportion is $2*xw+kw$. With a stronger round function, either the formula for the total number of rounds would be too complicated or performance would be hindered. So the aim was to find the simplest and the fastest function that takes no longer than that to build a PRF for blocks and keys of any size.

3.2 Search for the best round function

Intel assembly language includes LEA instructions that can shift a 32-bit word by 1, 2 or 3 bits and add to it another 32-bit word and a constant in a single instruction, thus providing multiplication by 3, 5 or 9 in a single adder. All the other shift-ADD combinations referred to as MUL [as the only considered multiplications] have to use an additional temporary register and a bitwise shift. Bitwise shifts are as slow as rotations but are not as strong. While constant shifts and rotations are almost free in hardware, they are the most expensive operations in software processors to be used in an ADD-XOR-ROL cipher. Therefore the use of shifts or rotations must be limited as much as possible. Unfortunately, that also has a limit: combining more than two different ADD-XOR or MUL-XOR operations per rotation shows no further improvement in strength.

Hundreds of different functions with all possible combinations of shifts, rotations and byte-swapping operations were tried. There was no time to research why some of them perform better than others. All the strong functions that allowed parallelisation 2-4 times such as (er2) $x_r \oplus= \mathrm{rotr}(x_{r-2} \oplus 2*x_{r\oplus 2} \oplus k_r \oplus r, w/4)*9 \oplus k_r$, were dropped after thorough testing, as their much slower diffusion rate makes up for all the gain from parallelisation but with a more complex round function that would not support 2-3-word blocks and would be faster only on some expensive processors. To achieve top performance, any change from the previous operation should be used in the following operation. The same applies to the round function when it is iterated in the opposite direction. So the best performing function would be of the form:

$$x_r = f(x_r, x_{r-1}, x_{r+1}),$$

of which the most efficient of the usable functions are of the form:

$$x_r \oplus= f\,((x_{r-1}<<c_1) \otimes (x_{r+1}<<c_2)),\ c_1 \neq c_2,$$

where f includes a rotation and $\otimes$ is a bitwise or arithmetic addition or subtraction. This is due to the fact that the register currently storing the last processed word in either direction gets updated with the output of the shift and addition operations and its original value is lost immediately. All other strong constructions require additional registers to implement.

After testing hundreds of thousands of similar variants, the following points became clear:

1) Of all the logical operations, XOR is the best choice. ADD provides enough nonlinearity.
2) Rotations work much better than shifts, but at the same cost in software.
3) It is essential to choose the right shift[s] and/or rotation[s].
4) A set of shifts and/or rotations works better than byte swapping or their combination.
5) At least two addition operations should be performed for every rotation.
6) Constants in one form or another are essential.
7) The round index must be included to remove self-similarity. It can replace the constants.
8) Different rotations or shifts should be used for different neighbours.
9) $f\,(2^*x_{r-1} \otimes x_{r+1})$ and $f\,(x_{r-1} \otimes 2^*x_{r+1})$ are the strongest choices that work equally well.
10) They are also the only functions that work for the smallest 2-word blocks.
11) Continuous addition of $x_{r+\lfloor xw/2 \rfloor}$ values and releasing output after 2+ rounds is the only scheme we could find that is guaranteed to resist guess-and-determine attacks for any size.
12) The simpler the key schedule the faster it builds a PRF. Optimally, each key word should be added twice, before and after the rotation.

This construction works for 2-word blocks thanks to its processing both sides with a slight offset shifting the left side left by 1 bit to satisfy the point (8) above. It is also the reason why an 8-bit rotation works with this round function: x_{r-1} is thus effectively rotated right by 7 bits feeding forward w-1 bits. Since this addition only diffuses bits of different words, diffusion of bits within each word is also necessary. It was added in the form of multiplication by 9 described above. With the 3-bit shift, the adder thus provides diffusion of the two neighbours and of the key bits shifted/rotated by 0, 4, 5, 7 and 8 bits. Multiplying the sum by 9 *before* the rotation is actually slightly weaker. The nonlinearity provided by this arithmetic addition is sufficient. It allows all the other operations to be linear.

8-bit rotation is usually a very bad choice for ADD-XOR-ROL ciphers, even in combination with other rotations. But this particular construction seems to be a very lucky case: an 8-bit rotation here works on words of any size as well as almost any other rotation. Although the strongest choice would be rotation right by 11 bits, it is only marginally stronger and only for large blocks. Rotation left or right by 8 bits performs equally well on small blocks where it matters the most. Since rotation right by 8 bits is slightly stronger on very large blocks than rotation left [but at the same low cost on 8-bit processors], it is our final choice.

EnRUPT+ with arithmetic additions before the rotation instead of the linear ones was also considered: $x_r \oplus = \mathrm{rotr}(2^*x_{r-1}+x_{r+1}+k_r+r, w/4)^*9 \oplus k_r$. It is as secure as EnRUPT with the same number of rounds and it is faster on Intel processors (3.8 clocks per byte in stream modes), but it is considerably slower on small 8-bit processors and at least two times slower in the FPGA and ASIC hardware.

Thanks to the extensive use of automated cryptanalysis to compare different primitives, the discovered round function is very close to optimal, although it was not the goal of this project. Some of the EnRUPT performance had to be traded in favor of its simplicity. Compare the EnRUPT round function (er1) with the TEA, XTEA and XXTEA shown below:

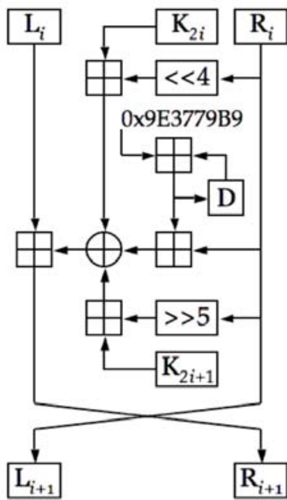

Fig 4. TEA

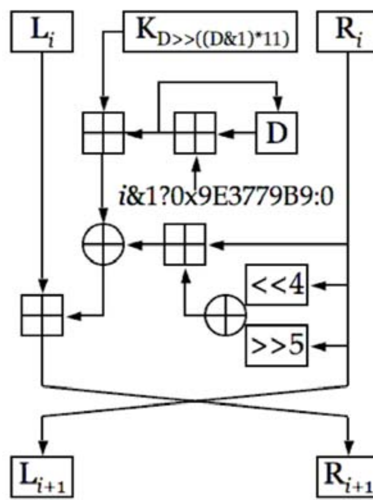

Fig 5. XTEA

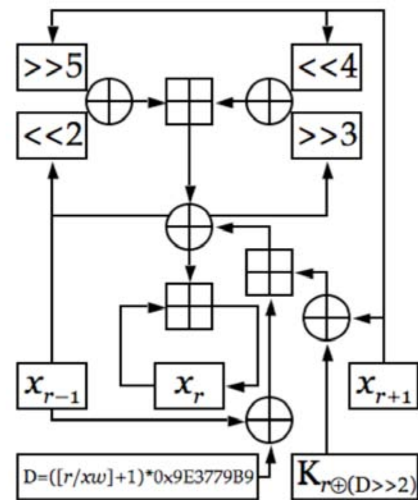

Fig 6. XXTEA

It should be quite obvious that EnRUPT round function is much smaller and much simpler than that of its closest relatives. It is also incomparably faster, especially on the 8-bit processors. We have designed EnRUPT aiming to replace its slower, more complex and limited predecessors.

3.3 Number of rounds

It is the hardest part of any cipher or hash function to get right. It is especially challenging to find the right formula for a fully scalable design like EnRUPT. And this is exactly where cryptologic theory ends and scientific guessing, guestimating and plain speculating usually begin. The only solid cryptologic base are the theoretical works pioneered by Luby and Rackoff [4] and improved and developed by Jacques Patarin [5], Naor and Reingold [6], Yun Park and Lee [7], Aiello and Venkatesan [8], Maurer [9], etc. These works provide theoretical proofs of security against various generic attacks for ciphers constructed with independent PRFs as round functions. Polynomial time distinguishers and independent PRFs are needed for these proofs of security to be applicable.

Approximately $2*xw+kw$ rounds of (er1) build a PRF for blocks and keys of any size with a controlled change anywhere in the data or key. The irreversible function (ir1) grows faster: approximately $xw+2$ rounds build a PRF for a state of any size with a controlled change in the input. Only the smallest sizes require an additional round or two. Those have a lower security margin.

Experimental results compared with the recent developments in cryptanalysis show that 4 times that number is the safest reasonable number of rounds for a secure cipher, which also seems to match Luby-Rackoff results. For example, mdRUPT-512-128-*x* similar to MD5 in proportions and structure of its round function [except (er1) is slightly stronger] is iterated for 96 rounds compared to the 64 rounds of MD5, and mdRUPT-512-160-*x* similar to SHA [except (er1) is stronger again] is iterated for 104 rounds compared to the 80 rounds of SHA. These numbers are much more reasonable than the 64 and 80 rounds chosen by the authors of MD5 and SHA. So far no one can claim that they can break 96 rounds of MD5 or 104 rounds of SHA. EnRUPT also demonstrates a much higher resistance to differential and rotational related key attacks[11] than the TEA, XTEA or XXTEA, which is essential if it is to be used as a hash function.

Cipher	PRF rounds (data/IV)	PRF rounds (key)	PRF rounds (both)	Proposed rounds	Maximum broken
TEA	10	∞	∞	64	∞[11]
XTEA	9	19	19	64	27[12] ≤ ?
XXTEA	10	15	21	64	?
RTEA	10	12	14	48	?
enRUPT-128-64	**8**	**10**	**10**	**32**	**–**
MD5-1	13	27	27	64	64 ≤ ?
MD5-2	11	25	25	64	64 ≤ ?
MD5-3	10	24	24	64	64 ≤ ?
MD5-4	13	26	26	64	64 ≤ ?
mdRUPT-512-128-*x*	**12**	**24**	**24**	**96**	**–**
SHA-0	16	27	27	80	80 ≤ ?
SHA-1	16	27	27	80	80 ≤ ?
mdRUPT-512-160-*x*	**14**	**25**	**25**	**104**	**–**
Salsa-20	4*16	4*16	4*16	20*16	7*16 ≤ ?
RUPT-512-512-*x*	**21**	**21**	**21**	**64**	**–**

Table 1. Strength of different primitives in PRF rounds

4. Security

EnRUPT security depends on the size of the data, key or hash. We recommend using keys, hash values and data blocks at least twice the security rating to resist parallel brute force and birthday attacks. Thus in general, we cannot claim that encryption or hashing by EnRUPT can provide more than the theoretical birthday bound of $w*min(kw,xw)/2$ bits of security, but in many applications such as block cipher constructions, EnRUPT can provide $w*sw$ bits of security with smaller ($sw \le kw < 2*sw$) keys without a nonce, with smaller ($xw < 2*sw$) blocks, or even with much smaller ($xw < sw$) blocks. A nonce or IV of the same size as the key concatenated with the key of the same size to form the kw words of the source block should also be sufficient to provide $w*kw$ bits of security in most applications.

Nobody cares what the code-maker has to say about security of his own design, but people do want to see reassurances that the best effort was made to ensure the cipher's resistance to all known and hopefully future attacks. By combining requirements for provably secure constructions set out in the fundamental theoretical works with automated cryptanalysis tools, EnRUPT will hopefully pave the road to automated cipher design. The requirements for provably secure constructions are the highest. Although a smaller number of rounds may resist known attacks, provably secure primitives require a much larger number of rounds. EnRUPT seems to be quite efficient even with such a large security margin enforced by the requirements for provable security.

We expect block EnRUPT variants with $s<4$ and $n<4*(2*xw+kw)-3$ rounds to be vulnerable to adaptive attacks (collision searches, boomerang attacks, etc.) and stream EnRUPT variants with $s<4$ and $n<4*(xw+2)-3$ rounds to be vulnerable to adaptive chosen IV attacks. We also expect block EnRUPT variants with $s<3$ and $n<3*(2*xw+kw)-2$ rounds and stream EnRUPT variants with $s<3$ and $n<3*(xw+2)-2$ to be vulnerable to non-adaptive attacks. Table 1 above shows that EnRUPT is also more resistant to related-key attacks than all of its closest relatives of any size.

All scalable ciphers such as EnRUPT or XXTEA updating w bits per round have another unusual side effect. Partial state collisions [two states identical except for a small part] can be found in arbitrarily large blocks after t full cycles with a total time*memory complexity of $2^{w*t/2}$, that is requiring a trade-off between 2^N time, $2^{w*t/2-N}$ memory and $\sim 2^{w*t/2-N}+2^N$ large block encryptions regardless of the round function. It is unknown at present if these partial state collisions pose any threat to security. The total time*memory*processors complexity of finding a partial state collision is 2^{96} for XXTEA and 2^{128} for EnRUPT. If resistance against partial state collisions in large target blocks [over 16 words in size] is required, block enRUPT and mdRUPT should be iterated for at least $\min(3*sw,\lfloor xw/2 \rfloor)*xw$ rounds. Partial state collisions have no effect on the stream EnRUPT variants even with very large states. The source code for the partial state collision search can be found at http://cryptolib.com/public/.

Since EnRUPT does not use variable rotations, S-boxes or other data-dependent or key-dependent memory operations, its software implementations are naturally resistant to timing attacks[13]. Hardware implementations, however, should take all the necessary precautions to ensure their resistance to side-channel attacks.

The cheapest generic attacks are parallel brute-force attacks. One block EnRUPT circuit is equivalent to about $w*(xw+kw+4)$ bits of memory. One stream EnRUPT circuit is equivalent to about $w*(xw+kw+5)$ bits of memory. EnRUPT updates one word of the state per clock cycle. Thus only the attacks requiring fewer clock cycles than would do as many EnRUPT circuits as can fit in the same area as the code, all the processors and all the memory required for the attack, can be considered faster or cheaper than the brute force.

4.1 Stream modes

It is possible that a stream cipher is always vulnerable to guess-and-determine (GD) attacks unless it updates more of its secret state than it releases on every round. Stream EnRUPT would be vulnerable to trivial GD attacks if it released one word on every round. If at least two rounds of EnRUPT are executed between outputs, then at least a half of the state must be guessed before any information can be gained from it. At such speed, $2*(xw+1)$ rounds are required to define the state. Since the attacker has no control over the state, it is also a sufficiently large state size/output proportion to resist passive statistical attacks. Many more rounds than that are required to over-define the state to perform algebraic attacks. The first SASC-2008 EnRUPT specification proposed sealing irRUPT state for n rounds after fast loading of xw words of input. Our collision resistance analysis has demonstrated this mode to be structurally insecure. Unkeyed unrandomized irRUPT should instead be iterated for $n=(s+2)*xw$ rounds while processing every xw input words. This updated specification reflects this correction. Only unkeyed unrandomized irRUPT mode is affected. Keyed or randomized irRUPT is identical to mcRUPT.

The average period of RUPT is $\sim 2^{w*((xw+1)/2+1)}$ words and the shortest guaranteed period is $2^{w+w/2}$ (2^{48}) words, thus applications that do not require more than $2^{w+w/2}$ (2^{48}) words of keystream can safely use smaller states of $xw=2*sw$ words, especially for CTR modes.

4.2 Hash function security

As a hash function, EnRUPT is required to provide first and second preimage resistance as well as collision resistance, which are ensured by its improved resistance to adaptive chosen/related key attacks thanks to the faster diffusion of key bits.

While irRUPT and mdRUPT-max are naturally invulnerable to length extension attacks, the fixed-block mdRUPT has many of the same weaknesses as any MD construction and therefore may need to be strengthened with such tweaks as prepending and/or appending the message length, randomization, etc. mdRUPT is included herewith only to provide a compatible immediate

replacement to the existing MD hashes and their implementations. The default mdRUPT variant is actually a strengthened MD construction: the recommended state size of the default mdRUPT is twice the size of the returned hash value and the recommended input size is half the state size thus turning mdRUPT into an expansion function compressing its state only during finalization.

Although (er1) compresses each keyword into 31 bits while processing each state word, it selects a differently compressed set of 31 bits of the key on every round expanding them to 32 bits of the state. Subsequent iterations load all other key bits, while control of each individual key word over the state word updated by it is limited just enough to prevent total cancellation of change by only one key word. Our tests show it to be a much stronger key schedule.

Other secure hashing modes can be easily used with enRUPT and irRUPT round functions. At this stage, no thorough collision resistance evaluation has been performed yet.

4.3 Traditional statistical attacks

Our initial linear and differential cryptanalysis showed absence of iterative characteristics in blocks of any size, but a detailed analysis of EnRUPT's resistance to all known statistical and algebraic attacks for blocks and keys of any size is a topic of further research.

5. Performance

Top performance was not the goal of this project. It was traded in favor of simplicity wherever necessary, but to a reasonable extent. Careful consideration was given to every possible choice and trade-off to build the most optimal and secure primitive.

5.1 Software performance estimates

According to our initial observations of its performance on Intel Core 2 Duo, different kinds of C implementations take 9 to 24 clock cycles per byte to hash/encrypt blocks of any reasonable but fixed size. Our flexible implementations supporting blocks of any size take 22 to 30 CPB. Stream RUPT, aeRUPT and mcRUPT take 3 to 7 CPB even in pure C. The unkeyed stream hash irRUPT is 3 times slower than that. Performance of simple implementations is very important since most software developers would prefer to avoid complex high-maintenance code, especially the kind they cannot understand. Unrolled EnRUPT-128 encrypts at about the same speed as the AES. These are not the best possible performance figures of course. EnRUPT can be implemented much better.

EnRUPT round function itself does not rely on the pipelining or scalar operations not available in the embedded processors, but it is still quite fast on all the modern processors outperforming many block and stream ciphers with a significantly smaller code that needs no additional memory. Pipeline optimisation is left to the compiler. Certainly, a parallel design exploiting vector operations and multiple pipelines can be significantly faster on modern processors, but we had to sacrifice some of the speed for simplicity and scalability to let the developers choose their own size and security parameters. Entire network packets, disk sectors or database records can be hashed and encrypted by the same code as single blocks, which reduces both processing and code maintenance overheads.

Most modern compilers will optimise out multiplication by 9 as addition of the same word shifted left by 3 bits or even as a single *lea eax*,[*eax**8+*eax*] instruction on Intel processors, but the more compact multiplication by 9 will hopefully also make it better suitable for the interpreted languages such as Java, JavaScript, PHP, Python, Basic or Perl. In the lower-level languages such as C or Pascal, it is better of course to implement EnRUPT with register operations either fully unrolled or as a small loop. To avoid confusing the less sophisticated compilers, the t*=9 operation can be implemented as t+=t<<3.

EnRUPT can also be implemented as a simple small macro or an inline function allowing it to be embedded into the code that can benefit from additional optimisations while also removing the overhead of function calls, which would be too expensive for a large cipher or hash function. This overhead is usually not taken into account by those who measure performance of ciphers expecting it to be always present. It is not the case with EnRUPT.

5.2 Embedded 8-bit processor performance estimates

EnRUPT is the first smartcard-friendly cipher with its tiny memoryless round function, a single **8-bit** or **16-bit** rotation and mostly XOR operations. Multiplication by 9 can be implemented as either a 3-bit shift across 32-64 bits and one 32-64-bit addition or as 4 calls to the adder implemented as 4-8 8-bit adders with carry. The rest of the cipher is trivially implemented with 8-bit XORs. Thus the total number of simple 8-bit operations to implement one round of enRUPT32/64 is 46/90. That is 17/33 XORs, 8/16 copy operations and either 21/41 arithmetic additions or 16/32 1-bit shifts and 5/9 arithmetic additions. It is 4 to 7 times faster than the AES-128 and 10 times faster than SHA-1, with a much smaller code and with lower memory requirements:

Primitive	RAM	Cycles per Block	Key Schedule Clocks
AES-128[14]	50	9464-13538	0-2278
enRUPT32-128	~38	~2208	0
enRUPT64-128	~42	~2160	0
SHA-1[14]	118	67244	478
mdRUPT32-512-320-160	~105	~6624	0
mdRUPT64-512-384-192	~113	~7200	0

Table 2. 8-bit processor performance comparison

5.3 Hardware performance estimates

EnRUPT round function needs much less than 1K logic gates plus $w*(xw+kw+1)$ flip-flops to store the key, the state and the round number. Stream EnRUPT variants occupy one additional word to store the feedback variable d.

At this stage, hardware performance of EnRUPT has not been fully studied yet. The area occupied by the EnRUPT round function is roughly w*(XOR4 + XOR3) gates + one w-bit carry adder. The two rounds of RUPT required to produce a single w-bit word of output have roughly the same logic depth as two w-bit carry adders. It means for instance logic depth of $1024/w$ carry adders to encrypt a 512-bit block with an initialisation overhead of $2176/w$ carry adders to load and seal a 64-bit IV into a keyed 512-bit state.

If ultra-high performance is required, RUPT should also be used in a pipelined CTR mode like any other cipher. Nothing is stopping the designers and developers from choosing the CTR mode for their protocols and for their products if the top speed is required at the cost of simplicity of the protocol and its implementations.

6. Further research

Performance of different software and hardware EnRUPT implementations needs to be investigated further. EnRUPT can be used with a large variety of the existing secure modes of operation. Performance of such modes of operation also needs to be investigated. Resistance of all EnRUPT variants to all known attacks and collision resistance of irRUPT and mdRUPT also need to be studied thoroughly before these primitives can be recommended for widespread use.

7. Acknowledgments

The authors express their highest gratitude to Karsten Nohl and to everyone who reviewed this paper for their invaluable notes and suggestions that helped us improve its quality.

8. Reference Implementation

```
// A fully unrolled C implementation of enRUPT-128 assuming 32-bit integers:

#define rupt1(x0,x1,x2,k,r) (x1^=rotr(2*x0^x2^k^r,8)*9^k)

#define enRUPT128(x0,x1,x2,x3,k0,k1,k2,k3) (\
rupt1(x0,x1,x2,k1, 1),rupt1(x1,x2,x3,k2, 2),rupt1(x2,x3,x0,k3, 3),rupt1(x3,x0,x1,k0, 4),\
rupt1(x0,x1,x2,k1, 5),rupt1(x1,x2,x3,k2, 6),rupt1(x2,x3,x0,k3, 7),rupt1(x3,x0,x1,k0, 8),\
rupt1(x0,x1,x2,k1, 9),rupt1(x1,x2,x3,k2,10),rupt1(x2,x3,x0,k3,11),rupt1(x3,x0,x1,k0,12),\
rupt1(x0,x1,x2,k1,13),rupt1(x1,x2,x3,k2,14),rupt1(x2,x3,x0,k3,15),rupt1(x3,x0,x1,k0,16),\
rupt1(x0,x1,x2,k1,17),rupt1(x1,x2,x3,k2,18),rupt1(x2,x3,x0,k3,19),rupt1(x3,x0,x1,k0,20),\
rupt1(x0,x1,x2,k1,21),rupt1(x1,x2,x3,k2,22),rupt1(x2,x3,x0,k3,23),rupt1(x3,x0,x1,k0,24),\
rupt1(x0,x1,x2,k1,25),rupt1(x1,x2,x3,k2,26),rupt1(x2,x3,x0,k3,27),rupt1(x3,x0,x1,k0,28),\
rupt1(x0,x1,x2,k1,29),rupt1(x1,x2,x3,k2,30),rupt1(x2,x3,x0,k3,31),rupt1(x3,x0,x1,k0,32),\
rupt1(x0,x1,x2,k1,33),rupt1(x1,x2,x3,k2,34),rupt1(x2,x3,x0,k3,35),rupt1(x3,x0,x1,k0,36),\
rupt1(x0,x1,x2,k1,37),rupt1(x1,x2,x3,k2,38),rupt1(x2,x3,x0,k3,39),rupt1(x3,x0,x1,k0,40),\
rupt1(x0,x1,x2,k1,41),rupt1(x1,x2,x3,k2,42),rupt1(x2,x3,x0,k3,43),rupt1(x3,x0,x1,k0,44),\
rupt1(x0,x1,x2,k1,45),rupt1(x1,x2,x3,k2,46),rupt1(x2,x3,x0,k3,47),rupt1(x3,x0,x1,k0,48))
```

The latest reference implementation of EnRUPT with test vectors can be found at http://www.enrupt.com/EnRUPT.zip

9. References

[1] D. J. Wheeler, R. M. Needham, “Correction to xtea”, Technical report, University of Cambridge, October 1998.

[2] B. Schneier, J. Kesley, “Unbalanced Feistel Networks and Block Cipher Design”, FSE-1996.

[3] S. O’Neil, “Algebraic Structure Defectoscopy”, Tools for Cryptanalysis 2007, Cryptology ePrint Archive, Report 2007/378.

[4] M. Luby, C. Rackoff, “How to construct pseudorandom permutations from pseudorandom functions”, SIAM J. Comput., vol.17, no.2, pp.373–386, April 1988.

[5] J. Patarin, A. Montreuil, “Benes and Butterfly Schemes Revisited”, ICISC 2005.

[6] M. Naor, O. Reingold, "On the Construction of Pseudo-Random Permutations: Luby-Rackoff Revisited", Journal of Cryptology: IACR 1996.

[7] A. Yun, J. H. Park, J. Lee, “Lai-Massey Scheme and Quasi-Feistel Networks”, Cryptology ePrint Archive, Report 2007/347.

[8] W. Aiello, R. Venkatesan, “Foiling Birthday Attacks in Length-Doubling Transformations”, EUROCRYPT 1996.

[9] U. M. Maurer, “A Simplified and Generalized Treatment of Luby-Rackoff Pseudorandom Permutation Generators”, EUROCRYPT 1992.

[10] D. J. Bernstein, “What output size resists collisions in a xor of independent expansions?”, cr.yp.to web site, May 2005.

[11] J. Kelsey, B. Schneier, D. Wagner, “Related-key cryptanalysis of 3-WAY, Biham-DES, CAST, DES-X, NewDES, RC2 and TEA”, Lecture Notes in CS, 1334: 233-246, 1997.

[12] Y. Ko, S. Hong, W. Lee, S. Lee, J. Lim, "Related key differential attacks on 27 rounds of XTEA and full rounds of GOST", FSE-2004.

[13] D. J. Bernstein, “Cache-timing attacks on AES”, cr.yp.to web site, November 2004.

[14] G. Keating, "Performance Analysis of AES candidates on the 6805 CPU core", Proceedings of The Second AES Candidate Conference, March 1999.