Hash family LUX - Algorithm Specifications and Supporting Documentation

Ivica Nikolić, Alex Biryukov, and Dmitry Khovratovich

University of Luxembourg

Abstract. In this paper we present the hash function LUX. It is a stream based hash design. In its internals LUX uses Rindael-like transformations. Hence, our security and efficiency margins are tightly related to those of Rijndael.

1 Introduction

Stream based hash function design is an alternative to the much more popular, and widely used, block based design. Typical examples of stream design are Panama[5], RadioGatun[3], Grindahl[13]. When the block based design usually operates with message blocks of 512 to 1024 bits of length, the stream design uses message blocks of much shorter length, say 24 to 64 bits. Other crucial difference between these two designs is the number of rounds spent to process one message block. For block design, usually a high number of rounds is used and each round is fairly simple. For example, SHA-1[8], a block based hash function, uses 80 rounds to process a message block of 512 bits. The stream based design on the other hand processes a small chunk of message in just one round, but this round has highly complex structure. Usually, a difference between these two designs is also noticeable in the size of their internal state, especially the number of registers (variables) used for the internal representation of the state. Block based functions have a relatively small number of registers in their internal state, e.g. SHA-1 has only 5 registers. But for stream based functions it is common for the internal state to be composed of a large number of registers.

In this paper we will present LUX, a stream based hash function. The design principle borrows from RadioGatun [3], yet the internal transformations are different: the core of these transformations is Rijndael based transformation. LUX scores high in both security and efficiency. Preliminary results show that LUX is resistant to the standard generic attacks (e.g. herding attack, multicollision attack, etc.) and has a high speed benchmark.

2 Specifications

LUX is a byte oriented stream based hash function. It supports 224, 256, 384 and 512 bit digests and can hash a message of length up to 2^{64} bits. The only parameter for the different digests is m which represents the size in the matrices

(the number of rows in the matrices) used for the internal representation of the state of the function. For LUX-224 and LUX-256 the value of m is 4, while for LUX-384 and LUX-512 it is 8.
LUX has an internal state S of $m \times 24$ bytes which can be divided into two parts:

- the buffer (B), which is a matrix of $m \times 16$ bytes, denoted $B_{i,j}$ where $0 \leq i \leq m-1$ and $0 \leq j \leq 15$
- the core (C), which is a matrix of $m \times 8$ bytes, denoted $C_{i,j}$ where $0 \leq i \leq m-1$ and $0 \leq j \leq 7$.

The internal state is manipulated by the state update function. The message is processed by small chunks of m bytes each.

Digest size	Message block (in bytes)	Core (in bytes)	Buffer (in bytes)	Internal state (in bytes)
224	4	4×8	4×16	96
256	4	4×8	4×16	96
384	8	8×8	8×16	192
512	8	8×8	8×16	192

Table 1. Parameters for different digests.

2.1 State Update Function

Important building block of LUX is the state update function Φ. It takes the current state S (buffer+core) and a message block M_t of m bytes and produces a new state, i.e.:

$$S_{new} \leftarrow \Phi(S_{old}, M_t).$$

This process is defined as *one round*. The state update function Φ itself can be decomposed into several consecutive steps. Let $B_{i,j} \in GF(2^8)$ be the elements of the buffer, and $C_{i,j} \in GF(2^8)$ be elements of the core. Let us denote by B_i the i-th column of the matrix B, i.e. $B_i = (B_{0,i}, B_{1,i}, \ldots, B_{m-1,i})^T$. Similarly, $C_i = (C_{0,i}, C_{1,i}, \ldots, C_{m-1,i})^T$. Let $M_t = (M_{m-1,t}, M_{m-2,t}, \ldots, M_{0,t})^T$ be the input message block of m bytes. Notice that message block is taken in reverse order. The reason for this is because LUX is little endian oriented function. Hence, when the message block of bytes is treated as a word on little endian architecture, no additional swapping of bytes is necessary. With "$\oplus$" we will denote vector XOR addition. Then the state update function, sequentially, does the following steps:

State update function Φ

Input $B = B_0||B_1||\ldots||B_{15} \quad C = C_0||C_1||\ldots||C_7$

1. **Add the message block to both the buffer and the core.**
   $B_0 \leftarrow B_0 \oplus M_t$
   $C_0 \leftarrow C_0 \oplus M_t$
2. **Update the buffer and the core separately.**
   $B \leftarrow F(B)$
   $C \leftarrow G(C)$
3. **Add the core to the buffer.**
   for $i = 0$ to 7 do
   $B_{i+4} \leftarrow B_{i+4} \oplus C_i$
4. **Feedforward column of the buffer to the core**.
   $C_7 \leftarrow C_7 \oplus B_{15}$

Output B,C

In the step 2 of Φ we use functions that transform the buffer and the core. Let us define these remaining functions F and G.
The function F is used to manipulate only the buffer B. It is a simple cyclic rotation of the matrix by one column to the right. Therefore $F(B)$, in pseudo code, can be defined as:

Buffer function F

Input $B = B_0||B_1||\ldots||B_{15}$

for $i = 0$ to 15 do
$\widetilde{B}_{(i+1)mod\ 16} \leftarrow B_i$

Output $\widetilde{B}$

The function G is one round of Rijndael where the core is seen as the state of Rijndael. This round transformation consists of SubBytes, ShiftRows, MixColumns, and AddConstant.

Core function G

Input C

$C \leftarrow \mathsf{SubBytes}(C)$
$C \leftarrow \mathsf{ShiftRows}(C)$
$C \leftarrow \mathsf{MixColumns}(C)$
$C \leftarrow \mathsf{AddConstant}(C)$

Output C

Although these four transformations are well known from Rijndael, below we will give a brief description of each of the transformations.
SubBytes. This non-linear byte-wise function is defined exactly as in the Rijndael specifications, i.e. the S-box used in SubBytes is the same as the S-box used in Rijndael.

$$S(X) = Y.$$

Let $X = X_1||X_2$, where X_1 are the first four bits of the byte X and X_2 the last four bits of X. Then the S-box used in LUX can be defined as:

$X_1 \backslash X_2$	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

ShiftRows. This transformation cyclically rotates to the left, independently, each row of the core matrix. For LUX-224 and LUX-256, whose core matrix has four rows, the rotation constants are 0,1,3, and 4. Let us denote this vector by ν, i.e. $\nu = (\nu_1, \nu_2, \nu_3, \nu_4) = (0, 1, 3, 4)$[1]. Then ShiftRows can be defined as:

$$C_{i,j} \leftarrow C_{i,(j+\nu_j) mod\ 8}$$

LUX-384 and LUX-512 have core matrix with eight rows, hence the rotation vector has eight coordinates: $\nu = (0, 1, 2, 3, 4, 5, 6, 7)$.

MixColumns. The MixColumns operation processes each column of the core matrix independently. A column is treated as an m-element vector in $\mathrm{GF}(2^8)$ and is multiplied by a matrix in $\mathrm{GF}(2^8)$. The multiplication is performed modulo the irreducible polynomial[2] $m(x) = x^8 + x^4 + x^3 + x + 1$. For LUX-224 and LUX-256 the matrix that defines this linear transformation is the following[3]:

$$C_i^{\mathrm{new}} \leftarrow A \cdot C_i^{\mathrm{old}}; \quad A = \begin{pmatrix} 02\ 03\ 01\ 01 \\ 01\ 02\ 03\ 01 \\ 01\ 01\ 02\ 03 \\ 03\ 01\ 01\ 02 \end{pmatrix}$$

[1] As in Rijndael-256.
[2] As in Rijndael.
[3] As in Rijndael.

For LUX-384 and LUX-512 the following matrix[4] is used:

$$A = \begin{pmatrix} 01\ 04\ 01\ 01\ 02\ 0c\ 06\ 08 \\ 08\ 01\ 04\ 01\ 01\ 02\ 0c\ 06 \\ 06\ 08\ 01\ 04\ 01\ 01\ 02\ 0c \\ 0c\ 06\ 08\ 01\ 04\ 01\ 01\ 02 \\ 02\ 0c\ 06\ 08\ 01\ 04\ 01\ 01 \\ 01\ 02\ 0c\ 06\ 08\ 01\ 04\ 01 \\ 01\ 01\ 02\ 0c\ 06\ 08\ 01\ 04 \\ 04\ 01\ 01\ 02\ 0c\ 06\ 08\ 01 \end{pmatrix}$$

AddConstant. In hash functions usually constant addition is introduced in order to stop various slide attacks. Therefore all the constants are different. Yet, our big state prevents that kind of attack. Nevertheless, Rijndael transformation is very sensitive to symmetric inputs. If all the elements of the core matrix C are the same, then the previous three transformations will produce a new state with, again, all elements equal to each other. To destroy this unwanted property we use the fourth transformation AddConstant. It is a simple XOR of 0x2ad01c64 to the core column C_0 (in byte representation, for LUX-224 and LUX-256, this means 0x2a is added to $C_{0,0}$, 0xd0 to $C_{1,0}$, 0x1c to $C_{2,0}$, and 0x64 is added to $C_{3,0}$). This constant corresponds to the first 8 hexadecimal digits of e.

AddConstant

Input C

$C_0 \leftarrow C_0 \oplus 0\text{x}2ad01c64$

Output C

In case of LUX-384 and LUX-512 the same constant is added to the core column C_0. The high 4 bytes are considered to be equal to zero, hence $C_{0,0}, C_{1,0}, C_{2,0}$, and $C_{3,0}$ stay unchanged after this transformation.

2.2 Initializations and padding

The buffer and the core are initialized by setting all the elements equal to zero. The standard padding procedure is applied to the message,. First, the message is divided into message blocks of m bytes ($8m$ bits). If the length of the last message block is less than $8m$ bits then this block is padded with one, followed by number of zeros such that the block length becomes equal to $8m$. If the last block has exactly $8m$ bits, than additional block is introduced, with one followed by $8m-1$ zeros. For LUX-224 and LUX-256, at the end two additional 4-byte blocks, containing the binary expression of the length of the non-padded message, are created. For LUX-384 and LUX-512 only one 8-byte block with the message length is created. This padded message becomes the input message for the hash function.

[4] As in Grindahl-512.

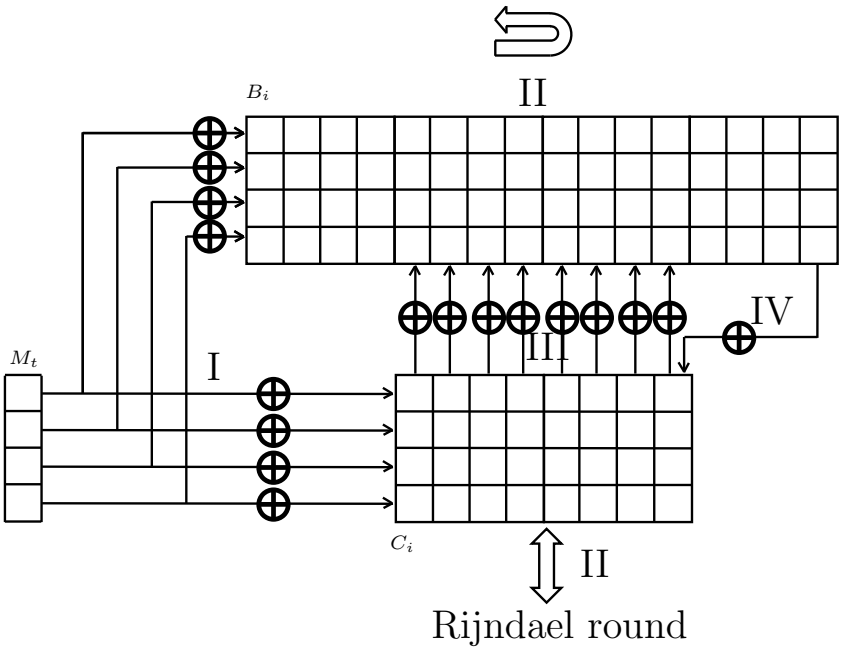

Figure 1: The round function of LUX. I - message addition to the buffer and the core. II - update of the buffer and the core. III - Addition of the core to the buffer. IV - addition of one column from the buffer to the core.

2.3 Hashing and output

The hash function LUX, similarly to the other stream based functions, uses the principle input message / blank rounds / output hash. These three phases are executed one after another. Depending on the message length and the size of the digest, the phases have different number of rounds.

Input phase After the state is being initialized and the message is being padded the input message phase starts. In this phase the whole message, block by block, is "absorbed", without outputting anything. Each message block, starting from the first one, is passed as an argument to the state update function Φ which transforms the internal state, in one round, to a new state. The input phase ends when, sequentially, all message blocks are processed. Obviously the number of rounds of this phase depends only on the size of the message, i.e. the number of message blocks in the padded message.

Blank rounds phase Blank rounds phase is typically used to increase the diffusion of the last message blocks. For LUX, this phase consists of 16 rounds. In each of these rounds the input message block is considered to be block with zero value.

Output phase In the last phase, the output hash phase, the hash value for the whole message is produced. In each round, the message input block, same as in the blank rounds phase, is a block with zeros. Yet, in each round m bytes of output are produced. The output of one round is the value of the core column C_3. Therefore LUX-224 produces its output in the following 7 rounds, LUX-256 in 8 rounds, LUX-384 in 6 rounds, and LUX-512 in 8 rounds. The output core column is treated in a little endian manner. Therefore first the lowest byte of this column is outputted, i.e. $C_{m-1,3}$, then the second lowest, i.e., $C_{m-2,3}$, then $C_{m-3,3}$,etc... The last output byte is the byte $C_{0,3}$.

From the algorithmic point of view, these three phases do not differ because they are divided into rounds and in each round the state update function gets the two necessary parameters: the message block and the internal state. Therefore the same function Φ can be used in all three phases. Let the padded message consists of k blocks. Then the hash function LUX can be defined as:

Hash function LUX-256

Input $M = M_0||M_1||\dots||M_{k-1}$

for i=0 to $k-1$
    $S \leftarrow \Phi(S, M_i)$
for i=1 to 16
    $S \leftarrow \Phi(S, 0)$
for i=1 to 8
    $S \leftarrow \Phi(S, 0)$
    print C_3

2.4 Reduced Variants

The block based hash function usually takes a high number of rounds to process one message block. By decreasing this number of rounds, reduced variants of the function are produced. In the analysis of the hash functions it is common to analyze these variants if the original function seems to be resistant to attacks. Since LUX uses different design principle (stream based), obviously, such variants are not possible. Yet, we can propose an alternative with decreased size of the internal state. Since the internal state of LUX consists of a core and a buffer, we propose a variants whose core and/or buffer have less number of columns. Although we claim it is critical to have a buffer that has 4 columns more than the core, in the front and at the end (in the passive buffer), yet it will be interesting to analyze how the security margin behaves for the variants with less number of columns in the buffer and the core. Reduced internal state variants of LUX-256 are proposed in Table 2.

Different variants can be made by reducing the number of blank rounds, from 16 rounds to 12 or 8 blank rounds. In the output hash phase in each round one column of output is produced. A variant, where in each round 2 or 4 columns of output are produced, can also be analyzed.

	Core	Buffer
LUX-256	4×8	4×16
LUX_{v1}-256	4×8	4×12
LUX_{v2}-256	4×4	4×16
LUX_{v3}-256	4×4	4×12

Table 2. Reduced variants of LUX-256.

We recommend these variants for the research on security margins of LUX but not for actual use.

2.5 HMAC, PRF, and Randomized Hashing

The LUX hash functions are suitable for a standard construction of HMAC:

$$\text{HMAC}_K(m) = h\big((K + c_1)||h((K + c_2)||m)\big).$$

The secret key K is padded with extra zeros to have a minimal length divisible by $4m$ ($8m$ if LUX-384 or LUX-512 is used). The constants have the following values: $c_1 = 0\text{x}5c5c \ldots 5c$, $c_2 = 0\text{x}3636 \ldots 36$. The inputs $K + c_1, K + c_2$ are divided into message blocks of m bytes and then processed by the hash function.

LUX can be used as a base of pseudo-random function (PRF) under different constructions, e.g., as HMAC-PRF.

Randomized hashing can easily be introduced in LUX by the following manner. After the initialization of the state, the first few rounds are used for randomized hashing. The input for these rounds is the value of salt. After the salting is finished, the hashing of the message starts and further processing is as described in the original hashing scheme. The minimal recommended salt size is $n/2$ bits for n bit digests. Hence, we propose salt of size 128 bits for LUX-224 and LUX-256. This salt is processed in $128 \div 32 = 4$ rounds. For LUX-384 and LUX-512 we propose 256 bit salt that is processed in $256 \div 64 = 4$ rounds.

3 Design Principles

LUX was designed with both security and efficiency in mind. We were impressed with the efficiency of the stream based hash designs Panama and RadioGatun [3, pp.12]. Yet, the updating functions of these designs had little publicity and analysis. Hence, we decided to utilize a well known and analyzed transformation. The best candidate, in our opinion, was Rijndael-like transformation. During the AES selection process Rijndael[7] was thoroughly analyzed and its security is well understood.

3.1 General Design Principles

The initialization with all-zero values was found to be practical and without any security flaws. The message padding with the length representation nowadays is a standard procedure in the design of modern hash functions. We use two additional blocks at the end for this representation of the message length because the size of our message block is limited to only m bytes. For LUX-224 and LUX-256, when $m = 4$, message block has 32 bits. In order to be able to hash messages of length up to 2^{64} we have to use two message blocks for the message length. The number of blank rounds was chosen such that the last message blocks travel through all stages of the buffer. This way the core can be influenced twice by the last message block, hence limit the possibilities of the attacker to manipulate only the core with the last message block.

3.2 Design of the Buffer

The simple linear buffer of Panama, based only on rotations, evolved to a more complex one in RadioGatun, by adding a feedback from the core to the buffer. This way it becomes harder to analyze the function because the feedback at the end of the buffer already depends on the previous states of the core. We have adopted this idea, so the buffer in LUX gets a feedback from the core.

It is easy to notice that the buffer is twice as big as the core: if the core is a matrix $m \times 8$ then the buffer is a matrix $m \times 16$. Regarding the buffer transformations, which can be either a rotation of the columns of the buffer, or addition the columns of the core to the columns of the buffer (feedback), the buffer itself, can be divided into two parts: 1) the middle part, the part with columns 5 to 12, where there is a rotation of columns and a feedback from the core, and 2) the passive part, the part with columns 1-4 and 13-16, where there are only column rotations. Increasing the number of columns in the part of the buffer that is passive has little influence on the efficiency because column rotations are costless. Yet, this increase has a big influence on the security of the function. We have chosen 4 columns in the front of the passive buffer because the message block that is added to the core gets a full diffusion after three rounds of Rijndael. One additional column (round) just increases this diffusion even more. A similar reasoning can be applied to the four columns at the end of the passive buffer.

3.3 Design of the Core

Our main arguments regarding the high security of LUX are based on the core transformation, which uses the Rijndael-like transformation. Various security and efficiency properties of Rijndael contributed to choosing it as our basic transformation. Our parameter m was chosen such that these properties can be brought to a forefront when implementing on 32-bit and 64-bit platforms.

We have chosen core of size $m \times 8$ (and not $m \times 4$) in order to decrease the ratio between the size of the message input, which is m bytes, and the size of

the core, which is $8m$ bytes. So, for LUX this ratio becomes 1/8 which provides better security margin. In Panama hash function, this ratio is around 1/2 and allowed to launch a practical attack. Suggestions were made by the designers of RadioGatun to decrease this value in order to limit the attacker's possibilities. In RadioGatun this ration is around 1/6. Notice, that we have adopted the suggestions made in the paper of RadioGatun, and count only the size of the core, without the size of the buffer.

The S-box used in LUX was chosen to be the same as in Rijndael. It has a maximum difference propagation probability of 2^{-6}.

The rotation vector in ShiftRows is optimal in a sense of ensuring that the whole state depends on each particular byte of the core as quickly as possible. All instances in the family LUX, require three rounds to get a full diffusion.

The matrices in the MixColumns were chosen so that it would have a maximal branch number. For LUX-224 and LUX-256 this is five, and for LUX-384 and LUX-512 it is nine.

4 Resistance to Attacks

Opposite to the widely used Merkle-Damgard construction, stream based hash functions had little publicity. Articles, where is done analysis of similar type of construction, Sponge functions [2],[4], are published. Yet, because of the blank rounds, LUX can not be considered a sponge function. For this reason, we can give preliminary results on the resistance to generic attacks of LUX-kind of constructions.

For LUX two types of collisions can be studied. One type are the collisions of the internal state before the blank rounds. This type of collisions are also known as *internal collisions*. Other type of collisions are the "traditional" *final collisions* of the hash values. Notice, that every internal collision can lead to final collision but not the opposite. The internal collisions are harder to achieve because of the larger internal state. Further we will analyze the security of LUX-256. Similar analysis (and conclusions) can be made for the rest of the functions in the family LUX. The internal state of LUX-256 has $24 \times 4 \times 8 = 768$ bits while it produces 256 bit output. Under the assumption that the underlying state update function is good, one needs, by the birthday paradox, $2^{\frac{768}{2}} = 2^{384}$ different internal states in order to find one internal collision. In the same time, for the final collision, this number is much smaller, $2^{\frac{256}{2}} = 2^{128}$. This property of having a internal state bigger than output hash length has proven to be very useful for increasing the complexity of some attacks. Benefits of a large state were pointed by Lucks in [14].

4.1 Generic Attacks

Most of the generic attacks in some way or another use internal collisions. These attacks applied to functions, whose size of the internal state is same as the output

hash size, can be very successful. Yet, by increasing the difference between the sizes of the internal state and the output hash, these attacks stop working.

LUX can be seen as a Merkle-Damgard construction with an output transformation. Each round of input message phase can be considered as one application of a compression function in the Merkle-Damgard model. The blank rounds and output hash phases are the output transformation in this model. So a collision in the compression function of Merkle-Damgard model means an internal collision for LUX. Therefore if one wants to launch a generic attack, applicable to the Merkle-Damgard construction that uses collisions, for the same attack on LUX he has to build internal collisions for LUX.

There are also a few attacks applicable to different spectrum of hash functions. Let us try to analyze the resistance of LUX to these attacks.

The Length Extension Property. A lot of hash functions that have an iterative structure suffer from the length extension property. For a function that has this property, once the attacker has one collision, he can easily build many other collisions. For example, for Merkle-Damgard functions, if the messages M_1 and M_2 collide then for any M the messages $M_1||M$ and $M_2||M$ also collide. LUX-256 has the length extension property, but it requires internal collisions to be found in the first place. This way, the first step, i.e. finding at least one collision, of the length extension attack requires 2^{384} effort.

Multicollision Attack. Multicollision set consists of messages that all hash to the same value. Efficient method was found by Joux [9], that can produce a multicollision set of 2^t messages with $t2^{n/2}$ effort for any n-bit Merkle-Damgard hash function. The brute force method for finding k multicollision set will take $2^{\frac{k-1}{k}n} \approx 2^n$. Joux's method is based on building consecutive collisions for the hash function using its iterative structure. For LUX these consecutive collisions all have to be internal collisions. Hence, the actual complexity depends on the size of the internal state. Since the internal state of LUX-256 has 768 bits, 2^t multicollision set can be built with $t2^{384}$ effort. This is higher than the simple 2^{256} brute force method which means LUX is resistant to multicollision attacks.

Herding Attack. In the Herding attack [10], the attacker presents the hash value of a message without knowing the beginning of the message. The main idea of the attack is building a so-called diamond structure: a binary tree of collisions. Similarly to the previous multicollision attack, in order to built the diamond structure, internal collisions have to be built. For Merkle-Damgard functions, the herding attack has complexity of around $2^{\frac{2n}{3}}$. But, for LUX, finding internal collisions depends on the size of the internal state, so one can expect that this attack complexity is $2^{\frac{2\times 768}{3}} = 2^{512}$. This number is much higher than the brute force attack on finding the required hash value which will take 2^{256} effort for LUX-256. Therefore LUX is resistant to the herding attack.

Second Preimage Attack. Kelsey and Schneier in [11], discovered a method that finds second preimages for long messages faster than the brute force. In one part of the attack they used the fact that having a set of 2^k intermediate hash values, one can expect to find a message which hashes to some value of this set with 2^{n-k} effort. These intermediate hash values for LUX mean internal state values, hence the probability of the attack increases greatly. The second part of Kelsey-Schenier attack, finding expandable messages, also deals with finding collisions which for LUX again means finding internal collisions. If the second preimage attack on n-bit Merkle-Damgard functions requires $2^{n/2} + 2^{n-k}$ for a message of 2^k blocks, then for LUX-256 the value of n can be replaces by the internal size, i.e. $n = 768$. Brute force second preimage attack requires around 2^{256} effort, which is much lower then the Kelsey-Schneier attack complexity. This means that LUX can resist the long message second preimage attack.

	Length extension	Multicollisions	Herding attack	Second preimage
ideal n-bit hash	immune	2^n	2^n	2^n
n-bit Merkle-Damgard	$2^{n/2}$	$t2^{n/2}$	$2^{2n/3}$	$2^{n/2} + 2^{n-k}$
LUX-n	$> 2^n$	2^n	2^n	2^n

Table 3. Resistance of LUX to generic attacks.

4.2 Dedicated Attacks

Differential trails. Most collision attacks on hash functions are based on the differential approach. A sequence of differences in internal states (*trails*) are built, and the messages are chosen such that the final difference is zero.

The differential trails where all differences are fixed values usually have extremely low probabilities in the case of byte-oriented hash functions. Each active S-box provides at maximum 2^{-6} multiplier to the full probability. As a result, one round of a trail with all S-boxes being active has probability lower than 2^{-256}.

The attack becomes more powerful if we consider trails with truncated differentials, where all non-zero byte differences are united in a single value. Since S-box converts zero difference to itself the S-box layer does not introduce any probability. However, the probability is provided by linear transformations (MixColumns in LUX).

Such an attack was successfully mounted on Grindahl by Peyrin in [15]. The best attack exploited a differential trail of probability 2^{-440}. Since parts of the trail can be attacked independently, the complexity of the full attack was only 2^{112} while the birthday bound is 2^{128}. In terms of conditions on byte differences, the original trail imposed 55 byte conditions, but 41 of them can be relaxed for free due to the freedom in the choice of message blocks at each injection.

Simultaneously, we have analyzed the resistance of LUX to this type of attack. The best attack we have found has complexity about 2^{400} and exploits a trail of probability around 2^{-704}. Again, the trail imposes 88 byte conditions, and 38 of them can be satisfied for free.

An extension to truncated differential attacks, the attack with structures[12], is faster but still not fast enough. The best trail for the structure attack provides an attack of complexity around 2^{256} for LUX-256, which is still much higher than the birthday attack.

Preimage attacks. The state update function is invertible, therefore, it is possible to compute backwards starting from any state. Nevertheless, starting from some state and reaching the initial state with all zero values in the registers, is supposed to be hard. Meet-in-the-middle attack, where the attacker starts from the initial state going forwards and from the final state going backwards, and try to to get the same intermediate value at some step, depends on the size of the internal state. LUX-224 and LUX-256 have an internal state of 768 bits, so meet-in-the-middle attack requires 2^{384} effort. This is much higher than the brute force preimage attack. LUX-384 and LUX-512 has a state of 1536 bits, so this attack would require 2^{768} effort. Again, simple brute force preimage attack has lower complexity.

4.3 Security levels

In addition to the security claims for generic attacks presented in Table 3, the following security levels, presented in Table 4, are claimed for the hash functions of the family LUX.

	Attack Complexity			
Digest	224	256	384	512
Collision attack	2^{112}	2^{128}	2^{192}	2^{256}
Preimage attack	2^{224}	2^{256}	2^{384}	2^{512}
Second preimage attack	2^{224}	2^{256}	2^{384}	2^{512}
HMAC-PRF distinguisher	2^{112}	2^{128}	2^{192}	2^{256}
Randomized hashing attack	2^{112}	2^{128}	2^{192}	2^{256}

Table 4. Security levels for LUX.

Also, any m-bit subset of the output bits, meets the required levels for collision resistance and preimage resistance of $2^{m/2}$ and 2^m respectively.

5 Performance

The main transformation implemented in LUX is the Rijndael transformation applied to the core. The transformation on the buffer, which is a simple rotation,

as well as the feedforwards have low computational complexities. Hence, it is to be expected that the performance of LUX is tightly related to the performance of Rijndael. Let us take a look at the software and hardware performance of LUX. The main point will be made on LUX-256 and LUX-224. Obviously they have the same efficiency. Then, the results and estimates will be extended for LUX-512 and LUX-384.

5.1 Software

It is well known from Rijndael's specifications that SubBytes, ShiftRows, MixColumns, and AddConstant transformations can be implemented altogether as simple table lookups and XORs [6, pp. 58-59]. Obviously, the same can be applied to LUX. In one round (one call of the state update function) LUX-256 uses 32 table look-ups and 25 XORs for the core transformation, 8 XORs for feedforward from the core to the buffer and 3 additional XORs: 2 XORs when adding the message to the core and the buffer, and 1 XOR for feedforward from the buffer to the core. So, in total, LUX uses 32 table look-ups and 36 XORs for processing 4 bytes of message. Regarding the memory requirements, LUX-256 needs 4KB for the tables and additional 96 bytes for the internal state. Hence, in total it requires little bit more than 4KB of memory.

Memory-speed trade-offs are possible. The entries of the look-up tables are rotated versions of one another. Therefore implementation with only one table and with total size (table+internal state) of 1096 bytes can be made. A variant with straightforward S-Box implementation would require only $256 + 96 = 352$ bytes of memory.

To get a sense of how fast LUX-256 is, let us try to compare its efficiency with AES (Rijndael-128). AES uses 4 table look-ups and 4 XORs per column per round. So, for processing 16 bytes, AES uses 160 table look-ups and 160 XORs. LUX, on the other hand, for processing 16 bytes (in 4 rounds) uses 128 table look-ups and 144 XORs. So the rough estimate is that LUX-256 is 1.2 times faster than AES. When comparing the efficiencies we have not included the blank rounds phase and output hash phase of LUX-256. Yet, these phases take only 24 rounds, a number insignificant for longer messages.

LUX-384, LUX-512. The longer digest variants of the family LUX are oriented towards 64-bit platforms. It can surely be implemented on 32-bit platforms but then the speed decreases about two times. For LUX-512 the same implementation tricks, as for LUX-256, can be used. Because of the bigger internal state, the precomputed tables are bigger. One table requires 2 KB and in total 8 tables are used. The internal state is 192 bytes. In total, for an optimal implementation, it is required around 16 KB of memory. The same memory-speed trade off can be made for LUX-512. Hence an implementation with only one table of 2KB is possible. A memory optimized implementation, without tables, would require $256 + 192 = 448$ bytes of memory.

The optimized implementation for the both, LUX-256 and LUX-512, takes 8KB + 16KB = 24 KB of memory. This amount of memory can easily fit into L1 cache of modern processors which have at least 32KB L1 data cache.

As already mentioned before, the initial value of LUX is all zero state. Hence, to set up the algorithm one needs 48 cycles: 16 cycles to set up the core and 32 cycles for the buffer.

	LUX-224	LUX-256	LUX-384	LUX-512
32-bit platform	16.7	16.7	28.2	28.2
64-bit platform	14.9	14.9	12.5	12.5

Table 5. Software estimates in cycles/byte for LUX with Intel Compiler 10.1. The 32-bit platform has Intel Core 2 Duo, 2.4 GHz clock speed, 2GB RAM. The 64-bit platform has Intel Core 2 Duo, 2.66GHz clock speed, 4GB RAM. Both platforms have Windows XP (x64) running.

In Table 5, cycles per byte estimate, obtained on our platforms, is presented. The implementations were made on C and compiled with Intel Compiler. Considering our previous estimate that LUX-256 is approximately 1.2 times faster than AES, and considering that the latest implementations of AES on Assembler [1] are reported to run at speed of 10 cycles/byte on some platforms, it is possible to expect that the speed of LUX will significantly increase with an optimized Assembler implementation.

8-bit Platforms. LUX is byte oriented, hence it can be easily implemented on 8-bit platforms. The S-Box can be implemented as table look-up. The multiplication of a variable with a constant, used in MixColumns, also can be implemented with XOR and table look-up. First a so-called $\mathsf{xtime}(x) = 02 \cdot x$ is implemented as table look-up. Then, all the multiplications with a constant can be implemented by xor-ing the results of a repeated use of xtime [6, pp. 54].

Our estimates for 8-bit platforms are presented in Table 6. A straightforward implementation is assumed, with two look-up tables of total size of 512 bytes for the S-Box and xtime and additional 96 bytes (192 bytes for LUX-384 and LUX-512) for the internal state.

	LUX-224	LUX-256	LUX-384	LUX-512
8-bit platform	190	190	190	190

Table 6. Estimates in cycles/byte for LUX on 8-bit Amtel AVR.

To set up the algorithm one needs to initialize the state with all zero bytes. Hence, for LUX-224 and LUX-256 the set up time is 192 cycles. For LUX-384 and LUX-512 it is 384 cycles.

5.2 Hardware

Hardware implementation parameters of LUX are also tightly related to those of AES. The framework of LUX-256 can be seen as Rijndael-256 with an additional component which is the buffer of LUX. Straightforward implementation of this buffer requires around 6K gates. Hence, the gate estimate of LUX-256 is around two times the estimate of AES plus additional gates for the buffer. The same reasoning can be applied to LUX-512 and get a factor of four for the gate count. A lightweight implementation of LUX-256 would require around 20K gates. For LUX-512 this number is slightly bigger, around 30K gates. A rough estimate for high-speed implementations is around 500K gates for LUX-256 and 1M gates for LUX-512.

6 Advantages and Limitations

The hash family LUX has the following advantages:

- It is based on Rijndael-like transformation. This is well known and thoroughly analyzed transformation. From point of view of security, the analysis of LUX can be focused only on the LUX construction: parameters of the core and the buffer and how they interact; there is no need to analyze the underlying transformation (because it is Rijndael). From point of view of efficiency, all implementation tricks of Rijndael can be used for LUX: fast implementation with only table look-ups and XORs, different time-memory trade offs, implementations for 8-bit smart cards and implementations in hardware. Moreover, any improvement in implementation of AES can directly influence in improvement for LUX.
- All major generic attacks are prevented by increasing the size of the internal state.
- Taking into account the results of [1] it is to be expected that LUX will perform at speed much higher than SHA-2.
- Digest sizes, other than the specified, can be supported. Actually, any digest size, less than 512 bits, can be produced by taking the values of the core column C_3 (possibly truncation of the last value of C_3 to some bits) in the output hash phase.

LUX is considered to have to following limitations:

- Producing digests for LUX-224 and LUX-256 takes almost the same amount of cycles. Same holds for LUX-384 and LUX-512.

– Short messages are hashed rather slow compared to long messages. This is because of the high number of rounds in the blank rounds and output hash phases which for LUX-256 and LUX-512 is 24 rounds. Yet, LUX has a size of a message blocks m bytes, where $m \in \{4, 8\}$. This fine parsing of the message in LUX, compared to the parsing of SHA where the block has a size of 64 bytes, decrease the padding size and partially compensates the underperformance for short messages.

References

1. Bernstein D.J., Schwabe P.: New AES software speed records. Cryptology ePrint Archive, Report 2008/381. See **http://eprint.iacr.org/2008/381.pdf**.
2. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: On the Indifferentiability of the Sponge Construction. In Smart, N. (ed.) EUROCRYPT 2008. (to appear)
3. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: RADIOGATUN, a belt-and-mill hash function. Presented at Second Cryptographic Hash Workshop, Santa Barbara, (August 24-25, 2006), See **http://radiogatun.noekeon.org/**
4. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Sponge Functions. Ecrypt Hash Workshop 2007, May 2007. Available at **http://sponge.noekeon.org/SpongeFunctions.pdf**
5. Daemen, J., Clap, C.S.K.:Fast hashing and stream encryption with PANAMA. In: Vaudenay, S. (ed.) FSE 1998. LNCS, vol. 1372, pp. 60-74. Springer, Heidelberg (1998)
6. Daemen, J., Rijmen, V.:The Design of Rijndael. Springer, Heidelberg(2002)
7. Daemen, J., Rijmen, V.: The Block Cipher Rijndael. In:Schneier B., Quisquater, J.-J (eds.) CARDIS 1998. LNCS, vol. 1820, pp. 277-284. Springer, Heidelberg (2000)
8. FIPS 180-1, Secure Hash Standard: Federal Information Processing Standards Publication 180-1, U.S. Department of Commerece/NIST, National Technical Information Service, Springfield, Virginia, Supersedes FIPS 180 (April 1995)
9. Joux, A.: Multicollisions in Iterated Hash Functions. In: Franklin, M. (ed.) CRYPTO 2004. LNCS, vol. 3152, pp.306-316. Springer, Heidelberg (2004)
10. Kelsey, J., Kohno, T.: Herding Hash Functions and the Nostradamus Attack. In: Vaudenay, S. (ed.) EUROCRYPT 2006. LNCS, vol. 4004, pp. 183-200. Springer, Heidelberg (2006)
11. Kelsey, J., Schneier, B.: Second Preimages on n-bit Hash Functions for Much Less than 2^n Work. In Cramer, R.J.F. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 474-490. Springer, Heidelberg (2005)
12. Khovratovich, D.: Cryptanalysis of Hash Functions with Structures. Hash Workshop, Leiden, 2-6 June 2008. Available at **lj.streamclub.ru/papers/hash/struct.pdf**
13. Knudsen, L.R., Rechberger, C., Thomsen, S.S.: Grindahl - A Family of Hash Functions. In: Biryukov, A. (ed.) FSE 2007. LNCS, vol. 4593, pp.39-57. Springer, Heidelberg (2007)
14. Lucks, S.: A Failure-Friendly Design Principle for Hash Functions. In: Roy, B. (ed.) ASIACRYPT 2005. LNCS, vol. 3788, pp. 474-494. Springer, Heidelberg(2005)
15. Peyrin, T.: Cryptanalysis of Grindahl. In: Kurosawa, K. (ed.) ASIACRYPT 2007. LNCS, vol. 4833, pp. 551-567. Springer, Heidelberg(2007)