Cryptanalysis of the Hash Function LUX-256

Shuang Wu Dengguo Feng Wenling Wu

State Key Lab of Information Security, Institute of Software
Chinese Academy of Sciences
Beijing 100190, China
Email: {wushuang,feng,wwl}@is.iscas.ac.cn

Abstract. LUX is a new hash function submitted to NIST's SHA-3 competition. In this paper, we found some non-random properties of LUX due to the weakness of origin shift vector. We also give reduced blank round collision attack, free-start collision attack and free-start preimage attack on LUX-256. The two collision attacks are trivial. The free-start preimage attack has complexity of about 2^{80} and requires negligible memory.

Key words : hash function, pseudo-random function, pseudo-collision, free-start preimage

1 Introduction

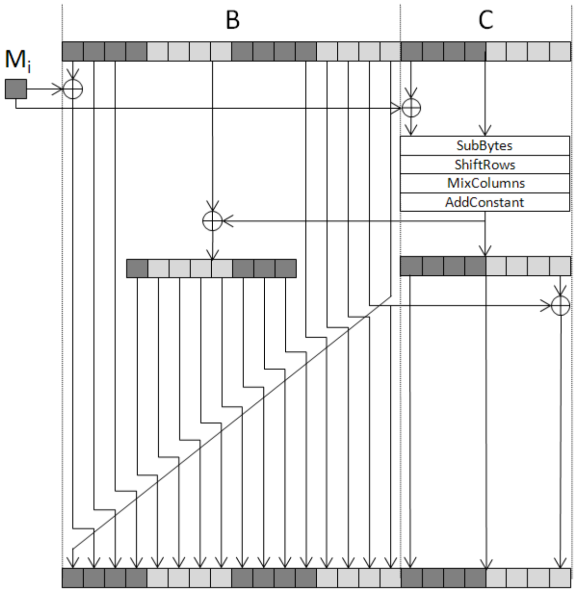
LUX is designed by Ivica Nikolić, Alex Biryukov, and Dmitry Khovratovich[1] which is submitted to NIST's SHA-3 competition. The stream-based structure of LUX is different from MD structure. There are several stream based hash functions: Radiogatun[4], Panama[5], Grindahl[6], Enrupt[2]. The structure based on stream can resist length extension attack, herding attack, multi-collision attack and meet-in-the-middle attack. But they also have weakness: the required buffer size is huge compared to MD structure and they can't be paralleled.

Section 2 of this paper is the description of LUX-256. Section 3 discussed some non-random properties of LUX-256. Section 4 and Section 5 introduced some collision attacks and a free-start preimage attack on LUX-256. Section 6 is the conclusion.

2 Description of LUX-256

LUX-256 has a buffer $B = (B_0, B_1, ..., B_{15})$ and a core $C = (C_0, C_1, ...C_7)$. B_i and C_i are 32-bit word. Initial values for B and C are zero. In each round a 32-bit message block is added to B_0 and C_0. After all message block are processed, there are 16 blank rounds. After the blank rounds, there are more blank output rounds. In each output round, the fourth byte C_3 of C is the output as a byte of the hash value. Output rounds continue until output length is enough. The state update operation of one round in LUX is shown in Figure 1.

Fig. 1. State Update Operation of LUX



The SubBytes, ShiftRows and MixColumns operations are AES-like. ShiftRows operation rotates cyclicly to the right and follows shift vector $(0, 1, 3, 4)$. S box used in SubBytes operations and matrix used in MixColumns operations are the same as in AES. AddConstant operation adds a constant $0x2ad01c64$ to C_0. If we use $\Phi(s, m)$ to denote one round operation of LUX, where m is a 32-bit message block and $s = B||C$ is the state value. LUX-256 can be written in pseudo-code as follows. For t message blocks,

```
s=0

For i=0 to t-1 do
$\Phi(s, m_i)$
End for

For i=0 to 15 do
$\Phi(s, 0)$
End for

For i=0 to 7 do
$\Phi(s, 0)$
output $s_{19}$    $(s_{19} = C_3)$
End for
```

3 Non-random Properties of LUX-256

3.1 Pseudo Random Distinguisher of LUX-256

Let $H : S \times M \to R$ be the hash function LUX and let $F : M \to R$ be a random function, where $M = \{0,1\}^*$ is the set of messages, S is the domain of initial state of LUX and R is the ranger of both H and F.

Take LUX-256 as an example, $S = \{0,1\}^{768}$ and $R = \{0,1\}^{256}$. The experiment of distinguishing LUX-256 from a random function F by adversary A is as follows:

```
Experiment $\mathbf{EXP}^{prf}_{H,A}$
$b \xleftarrow{R} \{0,1\}$
$F \xleftarrow{R} Rand^{M \to R}$
$K \xleftarrow{R} S$
$\mathcal{O}_0(\cdot) \leftarrow F(\cdot), \mathcal{O}_1(\cdot) \leftarrow H(K,\cdot)$
$d \leftarrow A^{O_b(\cdot)}$
return $d$
```

where F is randomly chosen from all functions mapping from M to R. K is randomly chosen from S, which is used as a secret key for H. Two oracle $\mathcal{O}_0(\cdot)$ and $\mathcal{O}_1(\cdot)$ are defined to be $F(\cdot)$ and $H(K,\cdot)$. A random bit b determines which oracle is provided to adversary A. And the adversary makes his guess bit d by querying the given oracle.

We give an adversary A_0 here:

```
Adversary $A_0^{\mathcal{O}(\cdot)}$
$m \xleftarrow{R} \{0,1\}^*$
$(h_0,h_1,h_2,h_3,h_4,h_5,h_6,h_7) \leftarrow \mathcal{O}(m)$
If $\mathcal{B}(h_i,h_{i+1}) = 1$ for all $i \in \{0,1,2,3,4,5,6\}$ return 1
else return 0
```

where m can be any message with arbitrary length. Returned value h is split into 8 32-bit word h_i. The Boolean function $\mathcal{B}$ is defined as:

```
Boolean Function $\mathcal{B}(V_0,V_1)$
$(a_0,b_0,c_0,d_0) \leftarrow V_0$,$(a_1,b_1,c_1,d_1) \leftarrow V_1$
If
    $f7 \cdot a_1 \oplus 4c \cdot b_1 \oplus f4 \cdot c_1 \oplus d_1 = 4e \cdot S(a_0)$
return 1 else return 0
```

where $S(\cdot)$ is the s-box used in AES and the calculations in the equations are all on $\mathbb{F}_{2^8}$ which are the same as in AES. The modular polynomial for multiplication is $x^8 + x^4 + x^3 + x + 1$.

Proposition 1. *The prf-advantage of LUX-256 is at least* $1 - 2^{-56}$.

Proof. Obviously, when $b = 0$*, for a random function* F*,*

$$P[\boldsymbol{EXP}^{prf}_{H,A_0} = 1|b = 0] = \prod_{i=0}^{6} P[\mathcal{B}(h_i, h_{i+1}) = 1] = \prod_{i=0}^{6} 2^{-8} = 2^{-56}$$

When $b = 1$*, it is the case of LUX-256, the output value is* $(h_0, h_1, h_2, h_3, h_4, h_5, h_6, h_7)$*. Split* h_i *into 8-bit strings* (a_i, b_i, c_i, d_i)*. Considering the MixColumn operations of the i-th word of output value, for all* $i \in \{0, 1, 2, 3, 4, 5, 6\}$*,*

$$\begin{pmatrix} 02\ 03\ 01\ 01 \\ 01\ 02\ 03\ 01 \\ 01\ 01\ 02\ 03 \\ 03\ 01\ 01\ 02 \end{pmatrix} \cdot \begin{pmatrix} S(a_i) \\ x_i \\ y_i \\ z_i \end{pmatrix} = \begin{pmatrix} a_{i+1} \\ b_{i+1} \\ c_{i+1} \\ d_{i+1} \end{pmatrix}$$

for some x_i, y_i, z_i*. We have,*

$$\begin{pmatrix} 02\ 03\ 01\ 01 \\ 01\ 02\ 03\ 01 \\ 01\ 01\ 02\ 03 \end{pmatrix} \cdot \begin{pmatrix} S(a_i) \\ x_i \\ y_i \\ z_i \end{pmatrix} = \begin{pmatrix} a_{i+1} \\ b_{i+1} \\ c_{i+1} \end{pmatrix} \tag{1}$$

and

$$\begin{pmatrix} 03\ 01\ 01\ 02 \end{pmatrix} \cdot \begin{pmatrix} S(a_i) \\ x_i \\ y_i \\ z_i \end{pmatrix} = d_{i+1} \tag{2}$$

From equation (1)*, we can calculate* (x, y, z) *as follows,*

$$\begin{pmatrix} x_i \\ y_i \\ z_i \end{pmatrix} = \begin{pmatrix} 03\ 01\ 01 \\ 02\ 02\ 01 \\ 01\ 02\ 03 \end{pmatrix}^{-1} \cdot \begin{pmatrix} a_{i+1} \oplus 02 \cdot S(a_i) \\ b_{i+1} \oplus S(a_i) \\ c_{i+1} \oplus S(a_i) \end{pmatrix} = \begin{pmatrix} f6\ 24\ 49 \\ f6\ 9f\ 24 \\ f6\ f6\ f6 \end{pmatrix} \cdot \begin{pmatrix} a_{i+1} \oplus 02 \cdot S(a_i) \\ b_{i+1} \oplus S(a_i) \\ c_{i+1} \oplus S(a_i) \end{pmatrix}$$

From equation (2)*, we have,*

$$d_{i+1} \oplus 03 \cdot S(a_i) = \begin{pmatrix} 01\ 01\ 02 \end{pmatrix} \cdot \begin{pmatrix} x_i \\ y_i \\ z_i \end{pmatrix} = \begin{pmatrix} f7\ 4c\ f4 \end{pmatrix} \cdot \begin{pmatrix} a_{i+1} \oplus 02 \cdot S(a_i) \\ b_{i+1} \oplus S(a_i) \\ c_{i+1} \oplus S(a_i) \end{pmatrix}$$

It turns out to be

$$f7 \cdot a_{i+1} \oplus 4c \cdot b_{i+1} \oplus f4 \cdot c_{i+1} \oplus d_{i+1} = 4e \cdot S(a_i)$$

So, $P[\mathcal{B}(h_i, h_{i+1}) = 1] = 1$ *for all* i*. Now we can see that*

$$P[\boldsymbol{EXP}^{prf}_{H,A_0} = 1|b = 1] = 1$$

which implies,

$$\boldsymbol{Adv}^{prf}_{H,A_0} = P[\boldsymbol{EXP}^{prf}_{H,A_0} = 1|b = 1] - P[\boldsymbol{EXP}^{prf}_{H,A_0} = 1|b = 0] = 1 - 2^{-56}$$

so,

$$\boldsymbol{Adv}^{prf}_{H} = \max_{A}\{\boldsymbol{Adv}^{prf}_{H,A}\} \geq \boldsymbol{Adv}^{prf}_{H,A_0} = 1 - 2^{-56}.$$

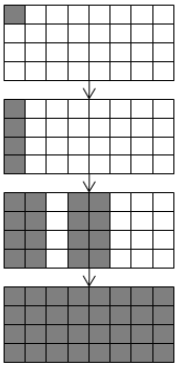
Clearly, LUX-256 is not a good pseudo-random function. The consecutive words in the hash value of LUX-256 have certain relations, which help our adversary to distinguish LUX-256 from a random function with only one query to the oracle. For other versions of LUX, this problem also exists. Especially for the versions whose core C is square, all shift vectors have zero in them.

3.2 Study of the Shift Vector

Our adversary works because there is a zero in this vector which implies that $S(a_i)$ involves in the calculation of $(a_{i+1}, b_{i+1}, c_{i+1}, d_{i+1})$ directly. In order to fix this problem, maybe a new shift vector should be found to replace $(0, 1, 3, 4)$. There should be no zero in the vector.

New vector should make sure the state in core C receives a full diffusion after three rounds and make sure the feedback block from B to C influence output part C_3 after two rounds as the origin vector $(0, 1, 3, 4)$ does. Figure 2 shows the diffusion process of one byte difference from message in several rounds.

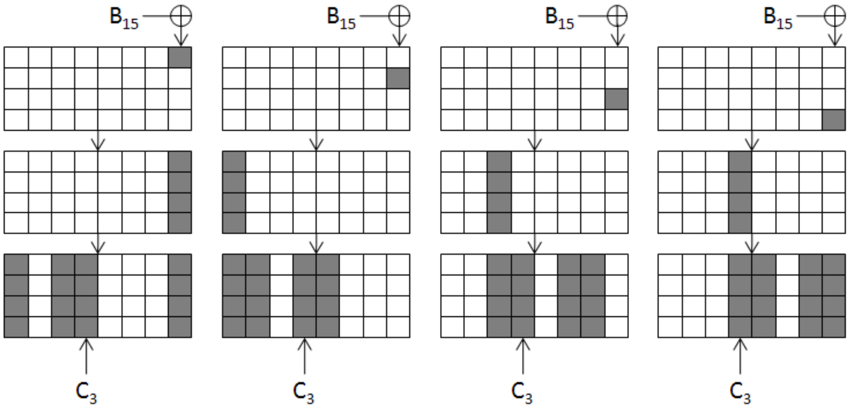
Fig. 2. Diffusion Process of LUX-256 with Origin Shift Vector



The feedback operation from B_{15} to C_7 can always influence the output part C_3 in at most two rounds with the origin vector as shown in Figure 3. New vector should do the same thing, or we can construct free-start near-collisions more efficiently and the civ-prf distinguisher's advantage will increase to $1 - 2^{72}$. We will talk about the civ-prf distinguisher in section 3.3.

Another property of the shift vector is each two of the value should be different. With all restrictions above, we can search for the new shift vector. It turns out that there are only 8 vectors which satisfy all conditions without consideration of order: $(1, 2, 3, 6)$, $(1, 2, 4, 5)$, $(1, 2, 6, 7)$, $(1, 4, 5, 6)$, $(2, 3, 4, 7)$, $(2, 3, 5, 6)$, $(2, 5, 6, 7)$, $(3, 4, 6, 7)$. Which of them is the best? We need more study to answer this question.

Fig. 3. Influence of Feedback Operation on Output with Origin Shift Vector



3.3 CIV-PRF Distinguisher of LUX-256

If the shift vector is changed to a new one without zero, the structure of LUX is still not of good randomness. In this section, we will show you a special distinguisher. If the adversary is given the right to chose initial state value instead of the message, he can still distinguish LUX-256 from a random function.

Let $H : S \times M \rightarrow R$ be the hash function LUX and let $F : S \rightarrow R$ be a random function, where we limit $M = \{0,1\}^{32}$ to be only 32-bit. The reason of the limitation will be explained later. S and R follow the same definitions in section 3.1. The experiment of civ(chosen initial value)-prf distinguishing LUX-256 from a random function F by adversary A is as follows:

Experiment $\mathbf{EXP}_{H,A}^{civ-prf}$
$b \overset{R}{\leftarrow} \{0,1\}$
$F \overset{R}{\leftarrow} Rand^{S \rightarrow R}$
$K \overset{R}{\leftarrow} M = \{0,1\}^{32}$
$\mathcal{O}_0(\cdot) \leftarrow F(\cdot), \mathcal{O}_1(\cdot) \leftarrow H(\cdot, K)$
$d \leftarrow A^{\mathcal{O}_b(\cdot)}$
return d

We give an adversary A_1 here:

```
Adversary $A_1^{\mathcal{O}(\cdot)}$
$s_0 \leftarrow 0^{768}$,$s_1 \leftarrow 0^{511}||1||0^{256}$
$(h_0, h_1, h_2, h_3, h_4, h_5, h_6, h_7) \leftarrow \mathcal{O}(s_0)$
$(h'_0, h'_1, h'_2, h'_3, h'_4, h'_5, h'_6, h'_7) \leftarrow \mathcal{O}(s_1)$
$(\triangle a_0, \triangle b_0, \triangle c_0, \triangle d_0) \leftarrow h_0 \oplus h'_0$
$(\triangle a_1, \triangle b_1, \triangle c_1, \triangle d_1) \leftarrow h_1 \oplus h'_1$
If
        $03 \cdot \triangle a_0 = 06 \cdot \triangle b_0 = 06 \cdot \triangle c_0 = 02 \cdot \triangle d_0$
and
        $\begin{pmatrix} \triangle c_1 \\ \triangle d_1 \end{pmatrix} = \begin{pmatrix} f7 & f4 \\ f6 & f4 \end{pmatrix} \cdot \begin{pmatrix} \triangle a_1 \\ \triangle b_1 \end{pmatrix}$
return 1 else return 0
```

Proposition 2. *The civ-prf-advantage of LUX-256 is at least* $1 - 2^{-40}$.

Proof. Obviously, when $b = 0$*, for a random function* F*,*

$$P[\boldsymbol{EXP}_{H,A_1}^{civ-prf} = 1|b = 0] = (2^{-8})^3 \cdot (2^{-8})^2 = 2^{-40}$$

When $b = 1$*, it is the case of LUX-256. We use* $(B_0, B_1, ..., B_{14}, B_{15})$ *and* $(C_0, C_1, ..., C_6, C_7)$ *to denote the buffer* B *and the core* C*. So in the beginning, since* $s = B||C$*, with the choices of the adversary, we have a difference in only* B_{15}*, which is* $\triangle B_{15} = 1$*. Now let's track the propagation of the difference* $\triangle B_{15}$*. It will propagate to the first output byte* $h_0 = \beta$ *as shown in Table 1, but we don't know the value of* β*.*

Table 1. Differential Path for CIV-PRF Adversary

round	$\triangle B$	$\triangle C$
0	---- ---- ---- ---1	---- ----
1	1--- ---- ---- ----	---- ----
blank-1	-1-- ---- ---- ----	---- ----
blank-2	--1- ---- ---- ----	---- ----
...	...	...
blank-14	---- ---- ---- --1-	---- ----
blank-15	---- ---- ---- ---1	---- ---1
blank-16	1--- ---α ---- ----	---α ----
output-1	-1-- ---β ξ-$\delta\epsilon$ ----	---β γ-$\delta\epsilon$
output-2	--1- ???? ???? ϵ---	???θ ????

Let $\alpha \xrightarrow{S} \alpha' = (\alpha'_0, \alpha'_1, \alpha'_2, \alpha'_3)^T$*, which means difference* α *changes to* α' *after it went through S box, then from Mixcolumn operation we have,*

$$\beta = \begin{pmatrix} 02\ 03\ 01\ 01 \\ 01\ 02\ 03\ 01 \\ 01\ 01\ 02\ 03 \\ 03\ 01\ 01\ 02 \end{pmatrix} \cdot \begin{pmatrix} \omega_0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 02 \cdot \alpha'_0 \\ \alpha'_0 \\ \alpha'_0 \\ 03 \cdot \alpha'_0 \end{pmatrix}$$

So the difference in the first output byte follows the pattern of β. When $b = 1$, the adversary makes two queries and calculate $(\triangle a_0, \triangle b_0, \triangle c_0, \triangle d_0)$, the differences satisfy

$$03 \cdot \triangle a_0 = 06 \cdot \triangle b_0 = 06 \cdot \triangle c_0 = 02 \cdot \triangle d_0$$

with probability of 1.

Let $\beta \xrightarrow{S} \beta' = (\beta'_0, \beta'_1, \beta'_2, \beta'_3)^T$ and $\epsilon \xrightarrow{S} \epsilon' = (\epsilon'_0, \epsilon'_1, \epsilon'_2, \epsilon'_3)^T$. We have,

$$\begin{pmatrix} \triangle a_1 \\ \triangle b_1 \\ \triangle c_1 \\ \triangle d_1 \end{pmatrix} = \theta = \begin{pmatrix} 02\ 03\ 01\ 01 \\ 01\ 02\ 03\ 01 \\ 01\ 01\ 02\ 03 \\ 03\ 01\ 01\ 02 \end{pmatrix} \cdot \begin{pmatrix} \beta'_0 \\ 0 \\ 0 \\ \epsilon'_3 \end{pmatrix}$$

So,

$$\begin{pmatrix} \triangle c_1 \\ \triangle d_1 \end{pmatrix} = \begin{pmatrix} 01\ 03 \\ 03\ 02 \end{pmatrix} \cdot \begin{pmatrix} \beta'_0 \\ \epsilon'_3 \end{pmatrix} = \begin{pmatrix} 01\ 03 \\ 03\ 02 \end{pmatrix} \cdot \begin{pmatrix} 02\ 01 \\ 01\ 01 \end{pmatrix}^{-1} \cdot \begin{pmatrix} \triangle a_1 \\ \triangle b_1 \end{pmatrix} = \begin{pmatrix} f7\ f4 \\ f6\ f4 \end{pmatrix} \cdot \begin{pmatrix} \triangle a_1 \\ \triangle b_1 \end{pmatrix}$$

with probability of 1, too.

Now we can see that,

$$P[\boldsymbol{EXP}_{H,A_1}^{civ-prf} = 1 | b = 1] = 1$$

which implies,

$$\boldsymbol{Adv}_{H,A_1}^{civ-prf} = P[\boldsymbol{EXP}_{H,A_1}^{civ-prf} = 1 | b = 1] - P[\boldsymbol{EXP}_{H,A_1}^{civ-prf} = 1 | b = 0] = 1 - 2^{-40}$$

so,

$$\boldsymbol{Adv}_{H}^{civ-prf} = \max_{A}\{\boldsymbol{Adv}_{H,A}^{civ-prf}\} \geq \boldsymbol{Adv}_{H,A_1}^{civ-prf} = 1 - 2^{-40}.$$

You can see that if length of the message is less than 32 bit, we need only one round before the blank rounds, so the difference in the last bit of B will not receive full diffusion.

In order to fix this problem, we could add two more blank rounds or always use two more padding message blocks. In other word, if there are two more rounds before output, this attack would not work any more.

4 Collision Attacks on LUX-256

Apparently, LUX has very good collision resistance. Once the difference goes into middle part of the core C, it's out of control, which brings chaos to buffer B. Since we can only control the first and the last byte in C, there are two kinds of trivial collisions.

Table 2. Differential Path for Reduced Blank Round Collision

round	$\triangle m$	$\triangle B$	$\triangle C$
0		---- ---- ---- ----	---- ----
1	α	-α-- β--- ---- ----	β--- ----
2	β	$\beta\alpha$-- -β-- ---- ----	---- ----
blank-1		-$\beta\alpha$- --β- ---- ----	---- ----
blank-2		--$\beta\alpha$ ---β ---- ----	---- ----
blank-3		---β α--- β--- ----	---- ----
output-1		---- $\beta\alpha$-- -β-- ----	---- ----
output-2		---- -$\beta\alpha$- --β- ----	---- ----
output-3		---- --$\beta\alpha$ ---β ----	---- ----
output-4		---- ---β α--- β---	---- ----
output-5		---- ---- $\beta\alpha$-- -β--	---- ----
output-6		---- ---- -$\beta\alpha$- --β-	---- ----
output-7		---- ---- --$\beta\alpha$ ---β	---- ---β
output-8		β--- γ-$\delta\epsilon$ ---η α---	γ-$\delta\epsilon$ ---ζ

4.1 Reduced Blank Round Collision

In this section we will talk about the blank rounds. If there are not enough blank rounds we can easily construct collision messages. See Table 2 for the differential path.

where $\alpha = (\alpha_0, 0, 0, 0)$ has only difference in the first byte. We choose difference of the second message word to be β which is pre-calculated from α. Before the difference goes into C, we have several steps to go. If there are only 3 blank rounds, we have collision pairs for LUX-224 or a near collision for LUX-256. One more blank round, we have one less colliding byte in the near-collision pair.

Difference stays in the first byte of C after one round, due to the zero in the origin shift vector, which provide us a chance to offset it. So, if we use a new shift vector which has no zero, this attack will no longer work.

4.2 Free-start Collision

If we can choose initial value, we can easily construct free-start collision for LUX. See Table 3 for the differential path.

This is pretty much the same as reduced blank round collision. We introduce some differences in the initial value. After two rounds all differences have gone, then we move on to the blank rounds and output rounds and get a collision.

Also, if we change the shift vector to a new one without zero, this attack will no longer work.

5 Free-Start Preimage Attack on LUX-256

Notice that the state update operation of LUX is invertible. If we have all the values in C and in part of B as the state just before the output rounds and the

Table 3. Differential Path for Free-Start Collision

round	$\triangle m$	$\triangle B$	$\triangle C$
0		α--β ---- ---- ---β	---- ----
1	α	β--- ---- ---- ----	β--- ----
2	β	---- ---- ---- ----	---- ----
blank-1		---- ---- ---- ----	---- ----
...		...	...
output-1		---- ---- ---- ----	---- ----
...		...	...

output calculated from them turns out to be the one we choose, we can calculate backwards to the initial value to get free-start preimages.

The question is, given hash value, how to decide the value in the buffer and core to make sure the output is exactly what we want. If we remove buffer B and let all rounds be blank, the sequence $C^0 \rightarrow C^1 \rightarrow ... \rightarrow C^6 \rightarrow C^7$ is determined by any one of the C^i. But our problem is, what is the relation between $(C_3^0, C_3^1, ..., C_3^6, C_3^7)$ and C^0, where $(C_3^0, C_3^1, ..., C_3^6, C_3^7)$ equals the given hash value

$$(h_0, h_1, ..., h_6, h_7) = \begin{pmatrix} a_0 & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 \\ b_0 & b_1 & b_2 & b_3 & b_4 & b_5 & b_6 & b_7 \\ c_0 & c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & c_7 \\ d_0 & d_1 & d_2 & d_3 & d_4 & d_5 & d_6 & d_7 \end{pmatrix}$$

From section 3.1, we know there are certain relations between h_i and h_{i+1}, so we assume the given hash value follows the pattern. Now we will try to find C^0 which will generate part of the hash value we want.

First, randomly choose B^0 and C^0 and let C_3^0 be h_0. Now we can modify some bytes to make sure C_3^1, C_3^2 and the first byte of all C_3^i be the same as the hash value. This attack consists of three phases.

5.1 Phase I-C_3^1

In phase I, we will modify C_3^1 to h_1. The Mixcolumn operation of the second output round is,

$$\begin{pmatrix} a_1 \\ b_1 \\ c_1 \\ d_1 \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \cdot \begin{pmatrix} S(a_0) \\ S(b) \\ S(c) \\ S(d) \end{pmatrix}$$

where b, c and d are located in C^0 as shown in Table 4. We can call them as the control bytes for C_3^1. Solve the equation group above to get the value of $S(b)$, $S(c)$, $S(d)$. Since we assumed that the given hash value follows certain pattern, there will be no contradiction while solving a 3-variable equation group of four equations. Then use inverted S box to calculate b, c and d. This is an easy phase.

5.2 Phase II-C_3^2

Phase II is more complicated. The mixcolumn operation for the third output round is,

$$\begin{pmatrix} a_2 \\ b_2 \\ c_2 \\ d_2 \end{pmatrix} = \begin{pmatrix} 02\ 03\ 01\ 01 \\ 01\ 02\ 03\ 01 \\ 01\ 01\ 02\ 03 \\ 03\ 01\ 01\ 02 \end{pmatrix} \cdot \begin{pmatrix} S(a_1) \\ S(e) \\ S(f) \\ S(g) \end{pmatrix}$$

where e, f and g is located in C^1, which are also shown in appendix table. The value of e, f and g can also be easily found by solving the equation group, but we can modify them directly. We must find their control bytes first. Control bytes for e ,f and g are denoted as e_i ,f_i and g_i. Unfortunately, g_3 is located in $C_{3,3}^0 = d_0$ which can not be modified. But we can used the feedback operation to modify it.

First, we calculate C^1 from C^0 without input of message and feedback. Second, we calculate the value of e,f and g. Now our goal is to change e^{old}, f^{old} and g^{old} to e, f and g, which means we need to bring difference $(0, 0, f \oplus f^{old}, 0)$, $(0, e \oplus e^{old}, 0, 0)$ and $(0, 0, 0, g \oplus g^{old})$ to C_0^1, C_2^1 and C_7^1.

We have two equation groups,

$$\begin{pmatrix} 0 \\ 0 \\ f \oplus f^{old} \\ 0 \end{pmatrix} = \begin{pmatrix} 02\ 03\ 01\ 01 \\ 01\ 02\ 03\ 01 \\ 01\ 01\ 02\ 03 \\ 03\ 01\ 01\ 02 \end{pmatrix} \cdot \begin{pmatrix} S(f_0) \oplus S(f_0^{old}) \\ S(f_1) \oplus S(f_1^{old}) \\ S(f_2) \oplus S(f_2^{old}) \\ S(f_3) \oplus S(f_3^{old}) \end{pmatrix}$$

$$\begin{pmatrix} 0 \\ e \oplus e^{old} \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 02\ 03\ 01\ 01 \\ 01\ 02\ 03\ 01 \\ 01\ 01\ 02\ 03 \\ 03\ 01\ 01\ 02 \end{pmatrix} \cdot \begin{pmatrix} S(e_0) \oplus S(e_0^{old}) \\ S(e_1) \oplus S(e_1^{old}) \\ S(e_2) \oplus S(e_2^{old}) \\ S(e_3) \oplus S(e_3^{old}) \end{pmatrix}$$

Solve them to get the values for all control bytes e_i and f_i, then modify them to get exact value of e and f in $C_{2,1}^1$ and $C_{0,2}^1$.

At last, modify $B_{14}^0 = B_{14}^0 \oplus g \oplus g^{old}$ to get g in $C_{7,3}^1$.

5.3 Phase III-the first byte of all remaining C_3^i

Now we have what we want in C_3^0, C_3^1 and C_3^2, for the remaining C_3^i, we modify B_{14}^{i-1} to make one byte $C_{3,0}^i$ be the same as the given hash value. From B_{14}^{i-1}, we can only modify $C_{7,3}^i$. We modify them round by round like this.

For round $i + 1$, we calculate C^{i+1} from C^i without input and feedback to get an output of $h_{i+1} = (a_{i+1}^{old}, b_{i+1}^{old}, c_{i+1}^{old}, d_{i+1}^{old})^T$. We need difference $\triangle a_{i+1}$ to be $a_{i+1} \oplus a_{i+1}^{old}$, and

$$\begin{pmatrix} \triangle a_{i+1} \\ \triangle b_{i+1} \\ \triangle c_{i+1} \\ \triangle d_{i+1} \end{pmatrix} = \begin{pmatrix} 02\ 03\ 01\ 01 \\ 01\ 02\ 03\ 01 \\ 01\ 01\ 02\ 03 \\ 03\ 01\ 01\ 02 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 0 \\ \triangle B \end{pmatrix} = \begin{pmatrix} \triangle B \\ \triangle B \\ 03 \cdot \triangle B \\ 02 \cdot \triangle B \end{pmatrix}$$

So $\triangle B = \triangle a_{i+1} = a_{i+1} \oplus a_{i+1}^{old}$. We can modify $B_{14}^{i-1} = B_{14}^{i-1} \oplus a_{i+1} \oplus a_{i+1}^{old}$ to make a_{i+1} be the same as given hash value.

5.4 Complexity of this attack

By modification, we have changed 136 bits to the given value. We use brute force to match remaining bits. Since the consecutive output words of LUX have relations between each other, we only need probability of 2^{-16} for each remaining step. The complexity of this attack is $2^{16\times 5} = 2^{80}$ and requires a fixed size of memory which is negligible.

If a new shift vector is used to remove relations between the output words, we can still apply similar attack with a lower probability of 2^{-24} for each remaining step. The total complexity turns out to be 2^{120}.

6 Conclusion

In this paper, we have introduced a new kind of distinguishing experiment called *civ-prf* (chosen-initial-value pseudo-random function) and modification techniques for free-start preimage attack. We can say that:

- LUX is not a good PRF nor a good civ-PRF, which can be fixed by changing a shift vector and adding two more rounds before output.
- LUX has good collision resistance except for reduced blank round collision and free-start collision attakcs. Since the structure of LUX is based on stream, these attacks won't affect the collision resistance of LUX.
- LUX has good preimage resistance except for free-start preimage attack. Preimage attack might work by improving the modification techniques and using partial-matching meet-in-the-middle techniques.

Open problems:

- The origin shift vector is not good, we need a new one without zeros. Which of them is the best?
- We have a lot of unused degrees of freedom in C_0 during the modification. Is there any way to improve the modification techniques to control more bytes?
- The state size is huge, and that's the main problem for meet-in-the-middle techniques. Can we apply the partial-matching techniques used in [3] to LUX?

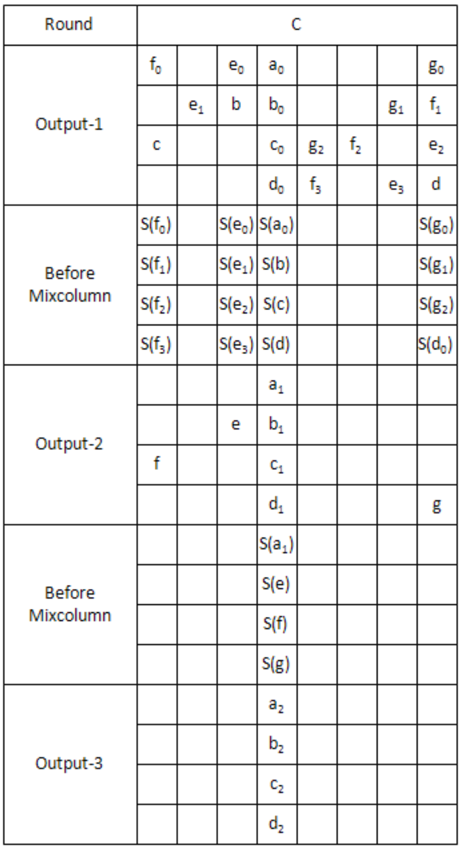
References

1. Ivica Nikolić, Alex Biryukov, and Dmitry Khovratovich. *Hash function family LUX*, 2008, submitted to the SHA-3 competition
2. Sean O'Neil, Karsten Nohl, and Luca Henzen. *EnRUPT hash function specification*, 2008, available at `http://enrupt.com/SHA3/`

3. Dmitry Khovratovich and Ivica Nikolić. *Cryptanalysis of EnRUPT*, 2008, available at `http://ehash.iaik.tugraz.at/uploads/9/9b/Enrupt.pdf`
4. Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. *Radiogatun, a belt-and-mill hash function*, 2006, available at `http://radiogatun.noekeon.org/`
5. Joan Daemen and Craig S. K. Clapp. *Fast hashing and stream encryption with PANAMA*. In FSE'98, volume 1372 of LNCS, pages 60-74. Springer, 1998.
6. Lars R. Knudsen, Christian Rechberger, and Søren S. Thomsen. *The Grindahl hash functions*. In FSE2007, volume 4593 of LNCS, pages 39-57. Springer, 2007.

Appendix

Table 4. Control Bytes For Modification Phase



Round	C							
Output-1	f_0		e_0	a_0				g_0
		e_1	b	b_0			g_1	f_1
	c			c_0	g_2	f_2		e_2
				d_0	f_3		e_3	d
Before Mixcolumn	$S(f_0)$		$S(e_0)$	$S(a_0)$				$S(g_0)$
	$S(f_1)$		$S(e_1)$	S(b)				$S(g_1)$
	$S(f_2)$		$S(e_2)$	S(c)				$S(g_2)$
	$S(f_3)$		$S(e_3)$	S(d)				$S(d_0)$
Output-2				a_1				
			e	b_1				
	f			c_1				
				d_1				g
Before Mixcolumn				$S(a_1)$				
				S(e)				
				S(f)				
				S(g)				
Output-3				a_2				
				b_2				
				c_2				
				d_2				