Collision Attack on Boole

Florian Mendel, Tomislav Nad, and Martin Schläffer

Institute for Applied Information Processing and Communications (IAIK)
Graz University of Technology, Inffeldgasse 16a, A-8010 Graz, Austria
Tomislav.Nad@iaik.tugraz.at

Abstract. Boole is a hash function designed by Gregory Rose and was submitted to the NIST Hash competition. It is a stream cipher based hash function which produces digests up to 512 bits. Different variants exist, namely Boole16, Boole32 and Boole64 where the number refers to word size in bits. Boole64 is considered as the official submission. In this paper we demonstrate a collision attack with complexity 2^{65} for the 64-bit variant and 2^{33} for the 32-bit variant. The amount of memory required is negligible. Since the attack on Boole32 is practical, we present an example for a collision.

1 Introduction

A hash functions maps an input of arbitrary finite length to an output of a fixed length. The basic security requirements for a cryptographic hash function are:

- *collision resistance* – it is computationally infeasible to find two different inputs, which hash to the same output.
- *second preimage resistance* – for a given input, it is computationally infeasible to find a second input with the same hash value.
- *preimage resistance* – for a given output of a hash function, it is computationally infeasible to find an input that hashes to that output.

Recently, the NIST hash function competition [1] started. In this public competition to find an alternative hash function to replace the SHA-1 and SHA-2 hash functions, many new designs have been proposed. In November 2008, round one has started and in total 51 algorithms were have been accepted. One of the submitted hash functions is Boole designed by Gregory Rose [2]. It is a stream cipher based design like PANAMA [3]. Boole is an expansion of the stream cipher Shannon [4] but is also influenced by other cryptographic primitives. Boole is a cryptographic primitive that can be used as a hash function, message authentication code (MAC) and a synchronous stream cipher.

In this paper we will describe a method to construct a collision for the Boole hash function. A collision occurs if two different messages result in the same hash value. Boole maps messages of arbitrary length to a hash result of 224, 256, 384 or 512 bits. A generic collision attack for the strongest version producing a 512 bit hash values requires about 2^{256} hash function computations. We will show that with our method a collision can be found with a complexity of less than 2^{65} state update transformations and negligible amount of memory.

M. Abdalla et al. (Eds.): ACNS 2009, LNCS 5536, pp. 369–381, 2009.

2 Description of Boole

Boole operates on W-bit words, $W \in \{16, 32, 64\}$. We refer to Boole16, Boole32 and Boole64 if we need to distinguish between the different word sizes. The Boole hash function supports output lengths up to $8 \cdot W$ bits. The internal memory consists of a 16-word register R and three word accumulators, namely x, r and l. The register is a nonlinear feedback shift register and at the end an output filter function is applied. Boole consist of three phases: input phase, mixing phase and output phase. In the following we explain these phases in more detail.

2.1 Input Phase

In the input phase, the accumulators and register words are updated with the message words m_t. Each message word is used once in the input phase.

$$\begin{aligned} temp &= f_1(l^{(t)}) \oplus m_t \\ l^{(t+1)} &= temp \lll 1 \\ x^{(t+1)} &= x^{(t)} \oplus m_t \\ r^{(t+1)} &= (r^{(t)} \oplus temp) \ggg 1 \\ R_3^{(t+1)} &= R_3^{(t)} \oplus l^{(t+1)} \\ R_{13}^{(t+1)} &= R_{13}^{(t)} \oplus r^{(t+1)} \end{aligned} \tag{1}$$

Afterwards the whole message has been processed and the register is cycled:

$$\begin{aligned} R_i^{(t+1)} &= R_{i+1}^{(t)}, \text{ for } i = 1, \cdots, 14 \\ R_{15}^{(t+1)} &= f_1(R_{12}^{(t)} \oplus R_{13}^{(t)}) \oplus (R_0^{(t)} \lll 1) \\ R_0^{(t+1)} &= R_1^{(t)} \oplus f_2(R_2^{(t+1)} \oplus R_{15}^{(t+1)}) \end{aligned} \tag{2}$$

In Figure 1 we have drafted the update step of the input phase.

2.2 Mixing Phase

After the input phase, the bit length of the input data, the output length and accumulators are mixed into the register. By *length* we denote the length of the input in bits, represented as a 64-bit integer and split into W-bit words. h is the length of the resulting hash value. The mixing phase is applied twice and is accomplished as follows:

$$\begin{aligned} R_0 &= R_0 \oplus length \\ R_4 &= R_4 \oplus l \oplus h \\ R_i &= R_i \oplus l, \forall i \in \{7, 10, 13\} \\ R_i &= R_i \oplus x, \forall i \in \{5, 8, 11, 14\} \\ R_i &= R_i \oplus r, \forall i \in \{6, 9, 12, 15\} \end{aligned}$$

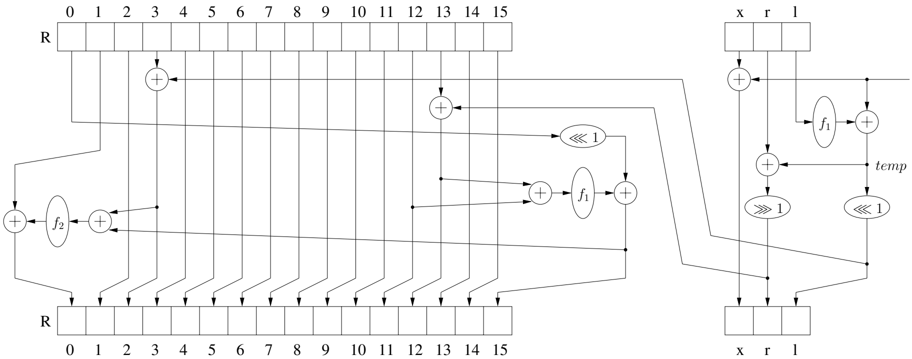

Fig. 1. Scheme of the update step

2.3 Output Phase

In the output phase, the content of the register and the output filter function is used to produce the hash value. First, the register is cycled as in Equation (2) and then, one word of the hash is computed as follows:

$$v = R_0 \oplus R_8 \oplus R_{12}$$

These steps are repeated until the required output length is reached.

2.4 Boolean Functions

The two nonlinear Boolean functions f_1 and f_2 depend on the the word size W. For Boole64 they are defined as follows:

$$\begin{aligned} t &= w \oplus \texttt{0x6996c53a} \\ t &= t \oplus ((t \lll C) \vee (t \lll D)) \\ t &= t \oplus ((t \lll B) \vee (t \lll E)) \\ t &= t \oplus ((t \ll A) \vee (t \lll F)) \end{aligned}$$

For $f_1(w) = t$ the constants $\{A, B, C, D, E, F\}$ are set to $\{3, 20, 34, 42, 55, 60\}$, and for $f_2(w) = t$ the constants $\{A, B, C, D, E, F\}$ are to $\{5, 27, 35, 46, 52, 55\}$. In the case of Boole32 the Boolean functions are defined as follows:

$$\begin{aligned} t &= t \oplus ((w \lll A) \vee (w \lll B)) \\ t &= t \oplus ((t \lll C) \vee (t \lll D)) \end{aligned}$$

For $f_1(w) = t$ the constants $\{A, B, C, D\}$ are $\{5, 7, 19, 22\}$ and for $f_2(w) = t$ the constants $\{A, B, C, D\}$ are $\{7, 22, 5, 19\}$. In Boole16, the Boolean functions are defined as follows:

$$\begin{aligned} t &= t \oplus ((w \lll A) \vee (w \lll B)) \\ t &= t \oplus ((\neg t \lll C) \vee (t \lll D)) \end{aligned}$$

For $f_1(w) = t$ the constants $\{A, B, C, D\}$ are $\{9, 13, 10, 15\}$ and for $f_2(w) = t$ the constants $\{A, B, C, D\}$ are $\{3, 14, 9, 10\}$.

3 A Differential Attack on Boole

In this section, we first analyze the differential properties of the components of Boole. We show that the Boolean functions f_1 and f_2 are not invertible and can be used to cancel differences. Then, we show how to find a collision in the accumulators and the register of Boole. Finally, we present a differential path which leads to a collision in the input phase. Since there are no message words used during the mixing and output phase, the collision in the input phase results in a collision of the full hash function Boole as well.

3.1 Collisions in the Boolean Functions

The Boolean functions f_1 and f_2 are used in every update step of the accumulator and the register of Boole. The main observation used in our attack is:

Observation 1. *The Boolean functions f_1 and f_2 are not invertible.*

Hence, we can find collisions in these functions and differences cancel out within the functions f_1 and f_2. In the following, we analyze which differences can be canceled and give the required conditions.

For Boole32 and Boole16 we get a zero output value for both f_1 and f_2 for the input values `0x0` and `0xF···F`. For Boole64 the input of the Boolean functions is first XORed with the constant `0x6996c53a`. Therefore, f_1 and f_2 collide for the values `0x6996c53a` and its inverted value `0x96693ac5`. The XOR difference for all variants of Boole is `0xF···F`. Note that there are more input values for f_1 and f_2 which collide. Table 1 shows all colliding input pairs with all-one difference for Boole32. Note that there are also more colliding input differences for the Boolean functions. However, in our attack we only use the all-one difference since this difference is rotation invariant and we can use the same difference in every step of Boole.

Table 1. Colliding input values for f_1 and f_2 with all-one difference for Boole32

w	$f_k(w)$	$f_k(w \oplus \texttt{0xFFFFFFFF})$
0x0	0x0	0x0
0x55555555	0x0	0x0
0xaaaaaaaa	0x0	0x0
0xFFFFFFFF	0x0	0x0

3.2 Difference Propagation in the Accumulator

In this section we show, how differences propagate and can be canceled in the accumulator. Whenever we injecting a message difference, we will first get a difference in all three accumulators x, r and l and the register words R_2 and

R_{12}. Remember that we can cancel the difference 0xF···F in the function f_1 of the accumulator. Hence, the shortest differential path which leads to a collision in the accumulator is by injecting the same message difference 0xF···F in two subsequent steps.

However, in this case the resulting differential path has a higher attack complexity. Therefore, we cancel the differences in the accumulator by injecting a second message difference after 3 steps. In this case, five differences are injected into the register.

The differences in the accumulators x and r are canceled by injecting the same difference in a subsequent message word. Whenever we inject the all-one difference using a message word, the resulting difference in the accumulator l is canceled using the function f_1 in the next step. According to Section 3.1, the difference 0xF···F cancels if the input value of f_1 (l_t) is 0x0 for Boole32 and Boole16 or 0x6996c53a for Boole64. If we inject the message difference 0xF···F in step t, the following equation needs to hold for Boole64:

$$l^{(t+1)} = f_1(l^{(t)}) \oplus m_t = \texttt{0x6996c53a} \tag{3}$$

Hence, the difference 0xF···F in m_t will cancel in the following function f_1 if the value of m_t equals:

$$m_t = f_1(l^{(t)}) \oplus \texttt{0x6996c53a} \tag{4}$$

3.3 The Differential Path

The full differential path, which leads to a collision in Boole is given in Table 2. Note that we only work with the all-one difference 0xF···F in the whole path. This has a number of advantages. First, we can and do always cancel the difference in the functions f_1 and f_2. Second, whenever two differences are XORed, the resulting difference is zero. This is especially useful in the XOR prior to the functions f_1 and f_2, since we do not need any condition in these cases.

We inject the first message difference in message word m_3 since we need the previous message words to fulfill the conditions on the following functions f_1 and f_2 (see Section 5). We inject two differences into the register and cancel the differences in the accumulator using the message word m_6. Afterwards we have five differences in the register. By canceling input differences for the Boolean functions, the five differences are moving through the register and after 16 cycles they are again at the same positions. By injecting the same differences in the message words Δm_{19} and Δm_{22}, the five differences in the register are canceled. Hence, we get a collision in the register, accumulators and the full hash function Boole after 23 update steps.

Figure 3 of Appendix A shows the beginning (step 3-7) and Figure 4 shows the end of the differential path (FF denotes the all-one difference). From these figures it is easy to see in which step we need to cancel differences in f_1 and f_2 by defining conditions on the input. The last column of Table 2 lists all occurring non-zero input differences in f_1 and f_2 of the register.

Table 2. Difference propagation. I and II are modifications to cancel a difference in f_1. III cancels a difference in f_2.

t	$R_0^{(t)}$	$R_1^{(t)}$	$R_2^{(t)}$	$R_3^{(t)}$	$R_4^{(t)}$	$R_5^{(t)}$	$R_6^{(t)}$	$R_7^{(t)}$	$R_8^{(t)}$	$R_9^{(t)}$	$R_{10}^{(t)}$	$R_{11}^{(t)}$	$R_{12}^{(t)}$	$R_{13}^{(t)}$	$R_{14}^{(t)}$	$R_{15}^{(t)}$	$x^{(t)}$	$r^{(t)}$	$l^{(t)}$	m_t	m.m. type
0																					
1																					
2																					
3																				Δm_3	I,III
4			FF										FF				FF	FF	FF		
5		FF										FF	FF				FF	FF			
6	FF										FF	FF	FF				FF	FF		Δm_6	I
7			FF							FF	FF	FF				FF	-	-	FF		
8		FF							FF	FF	FF				FF						
9	FF							FF	FF	FF				FF							II,III
10							FF	FF	FF				FF			FF					II
11						FF	FF	FF				FF			FF						
12					FF	FF	FF				FF			FF							II
13				FF	FF	FF				FF			FF								II,III
14			FF	FF	FF				FF			FF									III
15		FF	FF	FF				FF			FF										III
16	FF	FF	FF				FF			FF											III
17	FF	FF				FF			FF							FF					III
18	FF				FF			FF							FF	FF					III
19				FF			FF							FF	FF	FF				Δm_{19}	
20			-			FF							-	FF	FF		FF	FF	FF		
21					FF								-	FF			FF	FF			
22				FF									-				FF	FF		Δm_{22}	
23			-														-	-	FF		
24																					

4 Message Modification

In the this section, we explain how to modify the message words to get a zero output difference in Boole. Message modification was introduced by Wang *et al.* in [5]. The basic idea of message modification is to use the degrees of freedom one has on the choice of the message words to fulfill conditions on the state variables.

In our attack we distinguish between three different types of message modification, depending on how the conditions for the inputs f_1 and f_2 occur. Note that it is more difficult to fulfill the conditions, if they occur for both Boolean functions in the same step or if a message difference is introduced in the same step.

4.1 Type I Message Modification

This type covers the situation where a non-zero input difference for f_1 occurs and a message difference is injected in the same step. In that case, we have to adept a previous message word to get a zero output difference. Figure 2 shows how the previous message word influences the input of f_1 and we get the following message modification equations:

$$\begin{aligned} x &= ((m_{t-1} \oplus f_1(l^{(t-1)}) \oplus r^{(t-1)}) \ggg 1 \oplus f_1(l^{(t)}) \oplus m_t) \ggg 1 \oplus R_{13}^{(t)} \\ y &= (m_{t-1} \oplus f_1(l^{(t-1)}) \oplus r^{(t-1)}) \ggg 1 \oplus R_{13}^{(t-1)} \end{aligned} \tag{5}$$

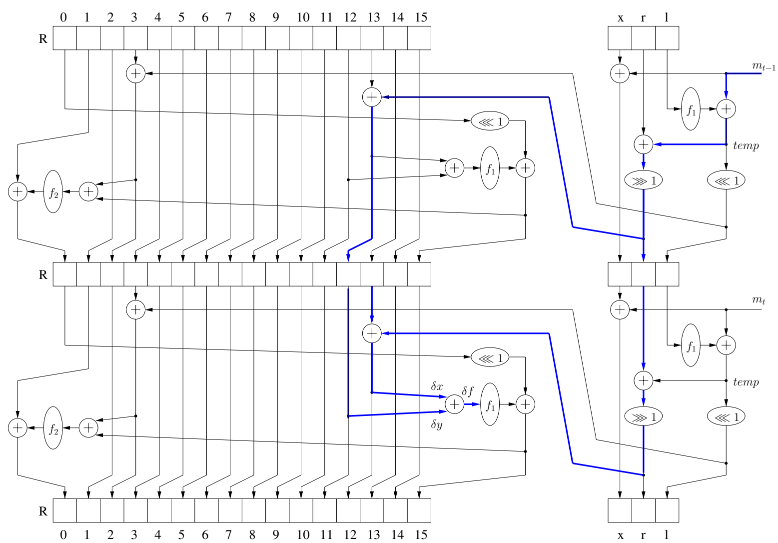

Fig. 2. Modification path for a collision in f_1

Hence, we have to find a message word m_{t-1} such that following equation holds:

$$x \oplus y = c,$$

where c is one of the values mentioned in Section 3.1. Instead of computing the message word itself, we compute the difference which is needed to change the current message word:

$$m_{t-1}^{\text{new}} = \delta m_{t-1} \oplus m_{t-1}$$

Then, equations (5) changes to

$$\begin{aligned}\delta x &= \delta m_{t-1} \ggg 2 \\ \delta y &= \delta m_{t-1} \ggg 1.\end{aligned}$$

Note that we ignore $f_1(l^{(t)}) \oplus m_t$, since it has always the same value, independent of the previous message words (see Section 3.2). We can then set up the following equation which expresses the needed difference for the input of f_1:

$$\delta f = \delta x \oplus \delta y = \delta m_{t-1} \ggg 2 \oplus \delta m_{t-1} \ggg 1 \tag{6}$$

For the value $c = 0$, δf is given by the following equation:

$$\delta f = R_{12}^{(t+1)} \oplus ((r^{(t)} \ggg 1) \oplus R_{13}^{(t)}) \tag{7}$$

Equation (6) defines a linear system of equations and $m_{t-1,j}$ denotes the jth bit of m_{t-1}:

$$\delta m_{t-1,i+1} = \delta m_{t-1,i} + \delta f_i \tag{8}$$

for $i = 0, \cdots W - 1$. To solve this system, we first choose a random value for $\delta m_{t-1,0}$. Then, we compute the remaining bits. Afterwards we check if the solution is correct by comparing

$$\delta m_{t-1,0} = \delta m_{t-1,W-1} + \delta f_{W-1}$$

to the randomly chosen value. A solution exists with probability 2^{-1}. If the solution is not correct we can choose a new message word m_{t-1}.

4.2 Type II Message Modification

The second case is much simpler and occurs if we have an input difference for f_1 in the register but we do not inject a message difference in the same step. Hence, we can achieve the needed input values for the Boolean function by modifying the message word in the same step t. The message is then computed as follows:

$$\begin{aligned}m_t &= (R_{12}^{(t)} \oplus R_{13}^{(t)}) \lll 1 \oplus r^{(t)} \oplus f_1(l^{(t)}) \\ m'_t &= m_t\end{aligned}$$

By this modification we get a zero output difference for f_1 with probability 1.

4.3 Type III Message Modification

For the case where a non-zero input difference for f_2 in step t occurs, we simply achieve a zero output difference by exhaustive search over all values of m_t or a previous message word, if in the same step also an other type of message modification has to be done.

5 The Collision Attack on Boole

In this section, all required steps to construct a collision for the Boole hash function, together with their complexities, are given.

1. The message words m_0, m_1 and m_2 are set to random values.
2. We inject a difference for m_3 and get a non-zero input difference for f_1 and f_2. We use type I message modification for f_1 and type III for f_2. Messages m_2 and m_0 are modified. The complexity of this step is 2^{W+1-d} update steps, where 2^d denotes the number of colliding input pairs for f_2 with all-one difference ($d = 2$ for Boole32).
3. Next we inject a difference in m_6 and get a condition for f_1. We solve this condition by type I modification of m_5. The complexity is about 2^1.
4. In step 9 we get again a non-zero input difference for f_2. A zero output difference is achieved by exhaustive search over all values of m_8. Additionally, a condition for f_1 is given which is solved by modifying m_9 according to type II message modification. The complexity of this step is 2^{W-d}.
5. In step 10 we get a non-zero input difference for f_1. We create a collision for f_1 by modifying m_{10} according to type II.
6. We do the same in step 12.
7. In step 13 we have conditions for f_1 and f_2. We do message modification of type II and III. For a zero output difference for f_2, m_{11} is used for the exhaustive search since m_{12} and m_{13} are already fixed. For each new value of m_{11}, m_{12} and m_{13} are recomputed. The complexity is again 2^{W-d}.
8. In step 14 we do an type II message modification of m_{14} to get a zero output difference for f_2. The same is done for step 15 and m_{15}, step 16 and m_{16}, step 17 and m_{17} and for step 18 and m_{18}. Each modification has a complexity of 2^{W-d}.
9. Finally, differences in m_{19} and m_{22} are injected which cancel all remaining differences.

The result is a collision in the register R and the accumulators x, r and l after the 23 step updates. Since all exhaustive searches are independent from each other, the overall attack complexity is given by $8 \cdot 2^{W-d} = 2^{W+3-d}$. For Boole64 this gives 2^{67-d} and we assume d to be at least 2. For Boole32 d is equal to two and therefore, the complexity is 2^{33} update steps.

5.1 Example Collision for Boole32

An example of two colliding message pairs for Boole32 is given in Table 3. The common hash value for both messages is

`3f71dd7bd86ac4731bc1567791d6fc8479c411530e3c8230d97cbca36c19e01f`.

Table 3. Two colliding messages for Boole32

m	a0bc0dbe	a1e5e09e	bcb01824	3403415f	0b177f21	7b31b82d	f5db2a23	a866bb7c
	004ebc0f	e11adc45	55b36c86	f59ed7ba	d7eb4405	c3265558	556eaf94	980d9839
	596fd2d9	d55ecff1	5df3155c	10dc14fa	22672d75	87fbd016	af0c15b8	4719bfdd
m'	a0bc0dbe	a1e5e09e	bcb01824	cbfcbea0	0b177f21	7b31b82d	0a24d5dc	a866bb7c
	004ebc0f	e11adc45	55b36c86	f59ed7ba	d7eb4405	c3265558	556eaf94	980d9839
	596fd2d9	d55ecff1	5df3155c	ef23eb05	22672d75	87fbd016	50f3ea47	4719bfdd

6 Conclusions

We presented a method to construct a collision for the Boole hash function. Boole was submitted to the NIST Hash competition, where the goal is to find a new secure hash algorithm (SHA-3). Boole is a stream cipher based design similar to PANAMA. However, we have shown in this paper, that Boole is not collision resistant. We are able to construct a collision in the internal register during the input phase. Since in the mixing and output phase no message inputs are used, this results in a collision for the whole hash function. In our attack we inject four message differences and have to modify a few messages words and after 23 steps the messages collide.

The main observation used in the attack is that the Boolean functions f_1 and f_2 are not invertible and we can construct collisions in these functions. The collision attack has a complexity of about 2^{W+3-d}, where W refers to the word size and 2^d the number of different colliding pairs for the Boolean functions f_1 and f_2. We provide an example of a colliding message pair for Boole32, since the attack complexity for this variant is about 2^{33} update steps and thus, feasible in practice.

Acknowledgements

The authors wish to thank Vincent Rijmen and the anonymous referees for useful comments and discussions. The work in this paper has been supported in part by the European Commission under contract ICT-2007-216646 (ECRYPT II) and by the Austrian Science Fund (FWF), project P19863.

References

1. National Institute of Standards and Technology: Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family. Federal Register Notice (November 2007), `http://csrc.nist.gov`
2. Rose, G.G.: Design and primitive specification for boole. Submission to NIST (2008), `http://seer-grog.net/BoolePaper.pdf`
3. Daemen, J., Clapp, 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)
4. Hawkes, P., McDonald, C., Paddon, M., Rose, G., de Vries, M.W.: Design and primitive specification for shannon. IACR EPrint Archive (2007), `http://eprint.iacr.org/2007/044`
5. Wang, X., Yu, H.: How to break md5 and other hash functions. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 19–35. Springer, Heidelberg (2005)

A Differential Path for Boole

On the following pages, we show the beginning and the end of the differential path, which leads to a collision in the hash function Boole.

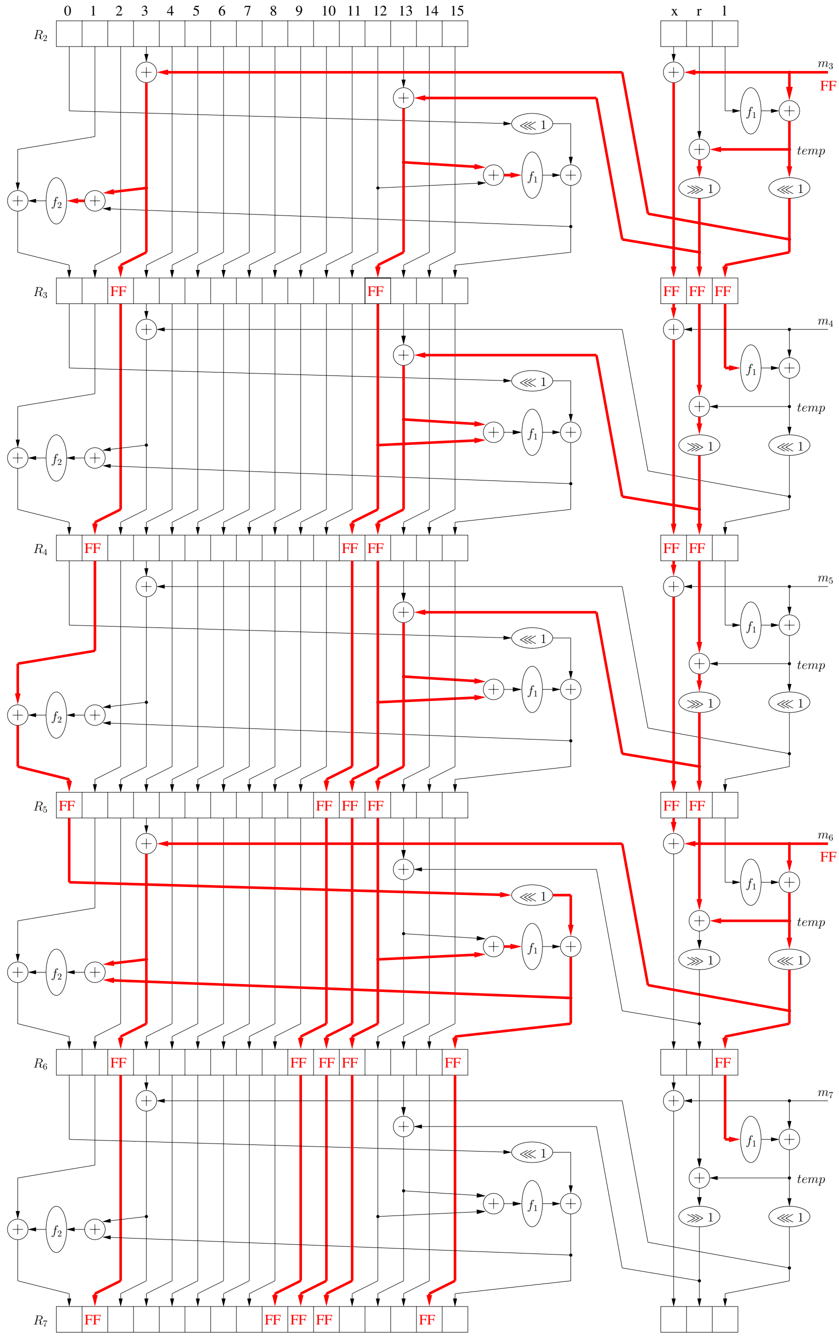

Fig. 3. Differential path for step 3 to 9

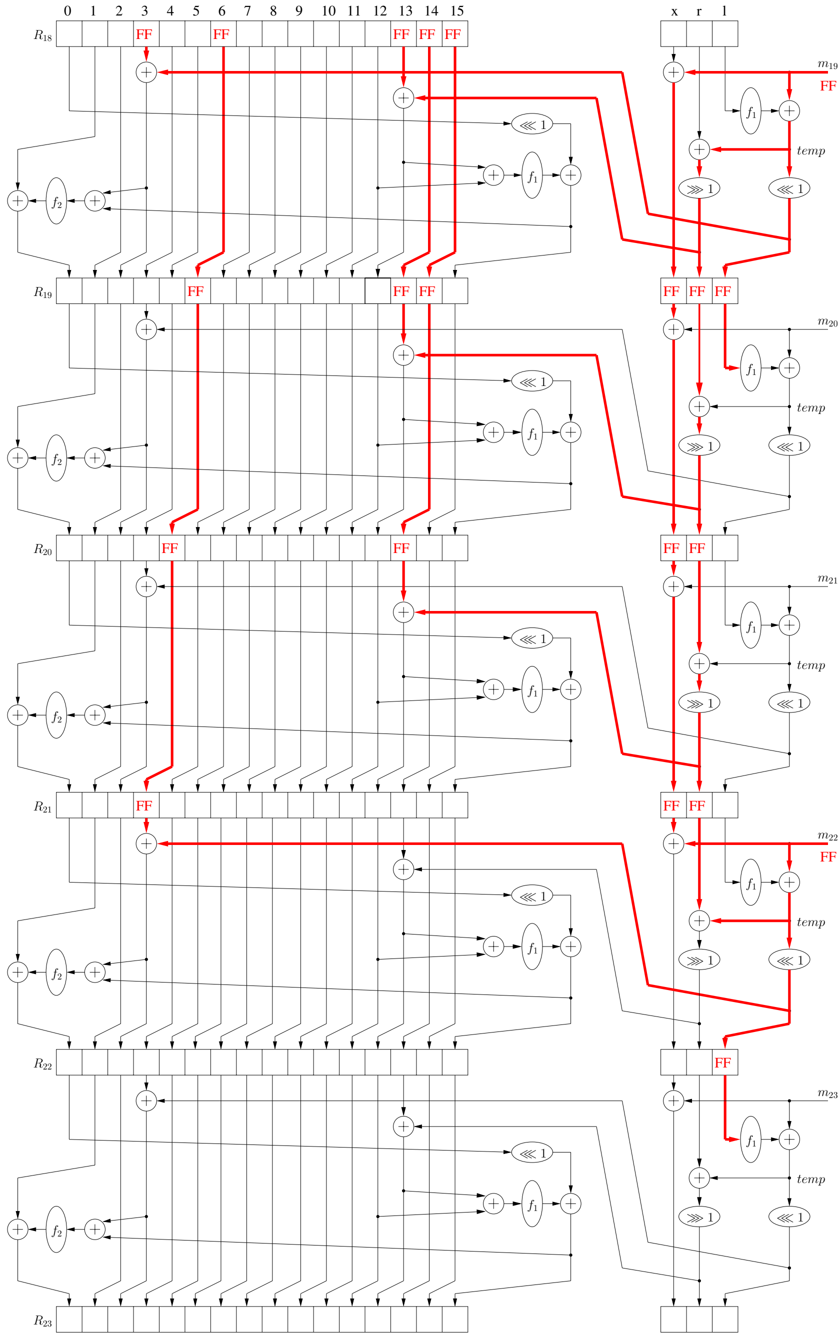

Fig. 4. Differential path for step 19 to 23