Attacking the Tav-128 Hash function

Ashish Kumar*⋆ and Somitra Kumar Sanadhya†

* Indian Institute of Technology, Kharagpur, WB, India
† Indraprastha Institute of Information Technology-Delhi, New Delhi, India
{kumarashish.iitkgp@gmail.com, somitra@iiitd.ac.in}

IIIT-Delhi Technical Report
Date: 28-July-2010

Abstract. Many RFID protocols use cryptographic hash functions for their security. The resource constrained nature of RFID systems forces the use of light weight cryptographic algorithms. Tav-128 is one such light weight hash function proposed by Peris-Lopez et al. for an RFID authentication protocol. In this article we show that Tav-128 is not collision resistant. We show a practical collision attack against Tav-128 and produce message pairs of arbitrary length which produce the same hash value under this hash function. We also study the constituent functions of Tav-128 and show that the concatenation of nonlinear functions A and B produces a 64-bit permutation from 32-bit messages. This could be a useful light weight primitive for future RFID protocols.

Keywords: Hash function, Tav-128, Cryptanalysis.

1 Introduction

RFID technology has gained wide acceptance in the market-place in the last decade. Due to the security vulnerabilities found in various RFID protocols and implementations, researchers have been focusing on designing secure RFID protocols. RFID tags have hard constraints on their size, the chip area and power consumption. Due to these constraints public key cryptography becomes generally impractical for use in the RFID protocols. Consequently, researchers have relied on cryptographic hash functions [6, 10, 9, 3, 2] or block ciphers [4] in the design of RFID protocols. Unlike block ciphers, hash functions do not require exchange of a key before operation and hence RFID community has thought them to be simpler to implement. However, Feldhofer and Rechberger [5] have shown

⋆ The present work was done while this author was doing Summer internship at IIIT-Delhi

that the number of gates required to implement most common hash functions on an RFID chip is much higher than that for some block ciphers. This has resulted in the development of lightweight hash functions which can be used in securing RFID protocols.

Tav-128 is a lightweight cryptographic hash function developed by Peris-Lopez et al. [8] which is utilized in an RFID authentication protocol. In this work we show that Tav-128 is not a strong hash function and practical collisions can be found for it for messages of any arbitrary length. The organization of the paper is as follows. In § 2, we present the notation used and describe the security requirements of a cryptographic hash function. In § 3 the structure of Tav-128 is explained. In § 4, we describe our collision attack and provide colliding message pairs for Tav-128. We conclude with some open problems on the constructions of light weight primitives for secure protocols in § 5.

2 Notation and Preliminaries

2.1 Notation

In this work, the following notation are used.
$+$: Addition modulo 2^{32}.
$-$: Subtraction modulo 2^{32}.
$||$: Concatenation of two quantities.
$\ll$: Left bit-shift operator (on 32-bit quantities).
$\gg$: Right bit-shift operator (on 32-bit quantities).

2.2 Security requirements of cryptographic hash functions

A cryptographic hash function produces a fixed length digest for an arbitrary sized message. It must satisfy some or more of the following properties, depending on the intended use of that hash function [7].

Pre-image Resistance: A hash function $h(.)$ is pre-image resistant if it is computationally infeasible to find an x for *any* given y such that $h(x) = y$.

Second Pre-image Resistance: A hash function $h(.)$ is second pre-image resistant if it is computationally infeasible to find an x_2 given *any* x_1 such that $h(x_1) = h(x_2)$ and $x_1 \neq x_2$.

Collision Resistance: A hash function $h(.)$ is collision resistant if it is computationally infeasible to find a pair (x_1, x_2), $x_1 \neq x_2$ such that $h(x_1) = h(x_2)$.

By "computational infeasibility", we mean that the complexity of an algorithm to break any one of the security properties is less than the generic attack for breaking that property. For a hash function producing an ℓ-bit digest, the complexity of birthday attack is $2^{\ell/2}$, and that for the pre-image and the second pre-image attack is 2^{ℓ}. If an attack can be described against any one of these properties and that attack has better complexity than these generic attacks then it is known as a "breaking" of the hash function.

In this work we attack the collision resistant of Tav-128 which produces 128-bit digests. Thus any attack requiring effort less than 2^{64} will be considered a break for Tav-128.

3 The Tav-128 hash function

As mentioned already, Tav-128 has been designed as a lightweight hash function to be used in an RFID authentication protocol [8]. The hash function outputs a digest of 128-bit for a message of any length. The structure of Tav-128 is as shown in Figure 1. Hashing for a message of length $32 \times k$ is done following the Merkle-Damgård structure where the compression function $f : \{0,1\}^{32} \times \{0,1\}^{160} \rightarrow \{0,1\}^{160}$ is iterated k times. The reference implementation of Tav-128 from [8] is provided in Appendix B.

3.1 The compression function f

Each call to the f function with a 32-bit message m updates 5 variables, each of which is of size 32 bits. These 5 variables are the register a_0 and the states $S[0], S[1], S[2]$ and $S[3]$. The f function utilizes 4 functions A, B, C and D to update these variables and also uses two internal variables h_0 and h_1 in this process. The schema of the compression function is described in Figure 2. The symbol $\oplus$ in this figure denotes bitwise XOR of two 32-bit quantities.

4 Collision attack on Tav-128

The compression function of Tav-128 first initializes the variables h_0 (resp. h_1) with a constant and then a non-linear function A (resp. B) updates this value depending on the 32-bit message m. The two functions A and B are shown in Table 1 below.

The authors of the hash function justify the inclusion of these two functions by stating that [8] "*We have also tried to include a filter phase*

Fig. 1. The Merkle-Damgård structure of Tav-128. Compression function f is $\{0,1\}^{32} \times \{0,1\}^{160} \to \{0,1\}^{160}$. Final transformation g simply outputs the final state S.

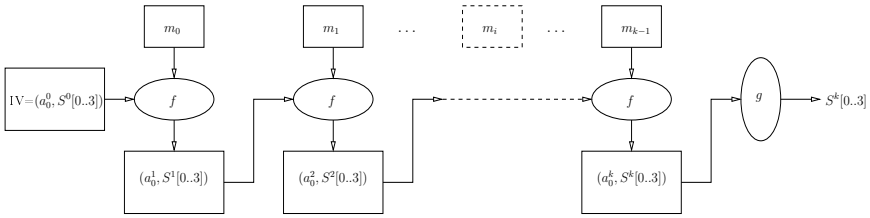

Fig. 2. Schema of the compression function f of Tav-128.

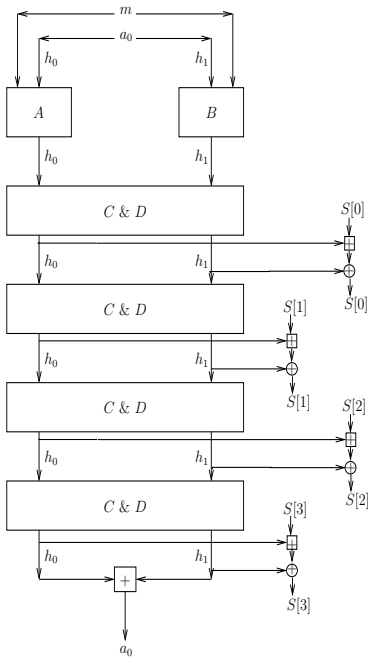

Table 1. Nonlinear functions A and B in Tav-128.

$A(h_0, m)$	$B(h_1, m)$
`for(i=0; i<32; i++)`	`for(i=0; i<32; i++)`
$h_0 = (h_0 \ll 1) + (h_0 + m) \gg 1$	$h_1 = (h_1 \gg 1) + (h_1 \ll 1) + h_1 + m$

(corresponding to algorithms A and B) in the input of the Tav-128 function, in order to avoid the attacker to have direct access to any bit of the internal state. Not having this possibility, some attacks that have been found on other cryptographic primitives in the past are precluded.".

In § 4.1, we investigate if the claim above is true and whether the application of these two functions weaken or strengthen the hash design.

4.1 Non-existence of collisions at the level of functions *A* and *B*

We note that the functions $A(h_0, \cdot)$ and $B(h_1, \cdot)$ are not permutations. It is easy to find collisions on m in either of the two functions A or B. Some such examples are presented in Table 2. Some more analysis of functions A and B is presented in Appendix A.

Table 2. Examples of collision in A and B functions.

S.	Function A			Function B		
No.	h_0	m	$A(h_0, m)$	h_1	m	$A(h_1, m)$
1	0x768c7e74	0x74093e01	0x6dabc1e3	0x768c7e74	0x4505b289	0x3d8fd817
		0x09057d79			0x62ee7bbd	
2	0x0	0x1bd18de3	0x099587a6	0x0	0x51b70ece	0xd3502587
		0x554216d3			0x17cac654	

Despite the fact that collisions in A and B are easily found, it does not appear easy to find collisions in *both* A and B *simultaneously*. Note that the only place where the message m is used in the compression function f is in functions A and B. Further computations in functions C and D only operate on the intermediate values h_0 and h_1 and state variables $S[0], S[1], S[2]$ and $S[3]$. Therefore, if a message pair m_1 and m_2 could be found such that $A(h_0, m_1) = A(h_0, m_2)$ and $B(h_1, m_1) = B(h_1, m_2)$ for the IV specified values $h_0 = h_1 =$ `0x768c7e74`, then the pair (m_1, m_2) will constitute a collision for the full hash function Tav-128. In order to find collisions in Tav-128, we therefore investigate collisions in the concatenation of outputs of functions A and B.

To search for collisions in $A(h_0, m)||B(h_1, m)$ we used the following strategy.

1. Create a table of size 2^{32} corresponding to all 32-bit messages m containing the triplet $(A(h_0, m), B(h_1, m), m)$.
2. Sort this table.

3. Look for a pair of adjacent rows in the sorted table where the first two entries are equal.

Since the size of the file in the strategy above will be of around 128 Gigabytes, sorting it would become computationally expensive on a standard PC. Therefore we modified the strategy slightly by dividing the file into 16 chunks of roughly equal size and then using disk sorting on each file of approximately 8 GB. Finally we combined these 16 files by using the merge sort algorithm. For sorting individual files, we used disk based sorting algorithm *psort* [1]. Our search reveals that there is *no* pair of messages on which both A and B functions collide starting from the IV. Thus there does not exist any collision on 32-bit messages at the level of A and B function.

4.2 On the map $(m||m) \rightarrow A(h_0, m)||B(h_1, m)$

As remarked in [8], individual functions A and B are quite efficient, requiring very few gates to implement them. Thus they seem to have potential applications in light weight protocols. However, our discussion in § 4.1 shows that it is very easy to find collisions in these individual functions. Therefore the use of these functions in any application where collisions in these functions could cause loss of security is immediately ruled out. Contrary to what one would expect, however, we have found that the 64-bit map $(m||m) \rightarrow A(h_0, m)||B(h_1, m)$ *is* a permutation. Since this is a light weight permutation, constructed from two light weight primitives, it may be a useful tool for future protocols requiring low cost constructs.

4.3 Finding collisions at the level of C and D functions

From Figure 2 we note that if the intermediate values h_0 and h_1 collide for two different messages just after one application of C and D functions, then $S[0]$ will have the same value for both the messages. Since the message does not get used in subsequent computations, all the register values (h_0, h_1, $S[0]i$, ..., $S[3]$) and hence the final hash output will also be same for these messages.

We used a strategy similar to the one described in § 4.1 to generate such message pairs. We obtained 11 message pairs which collide for Tav-128. Two of these pairs, the colliding hash value and the intermediate values of h_0 and h_1 after the fist application of C and D function are presented in Table 3 next.

The cost of the attack: The estimate of effort is about 2^{32} calls to about $1/4^{th}$ of the hash function, followed by sorting the lists of h_0 and h_1 and subsequent search for colliding pair. The whole process of creating the files, sorting them, merging to create one file and finally searching for colliding pairs on this merged file took less than 1 working day on a standard PC. We estimate the effort to be about 2^{37} calls of Tav-128. This is significantly below the birthday bound of 2^{64} for a hash function producing 128-bit digests.

4.4 Colliding message pairs

Two pairs of colliding messages are presented in Table 3. Since the chaining variables $h_0, h_1, S[0], S[1], S[2]$ and $S[3]$ are all equal for each message pair, appending any arbitrary message at the end of the colliding pair will still collide for Tav-128. Given the results in Table 3, it is trivial to construct messages of any length $\geq$ 32-bit which will collide for Tav-128.

Table 3. Colliding message pair for Tav-128. Both the message pairs are 32 bits and the hash output $H(M)$ is 128 bits. h_0 and h_1 are the intermediate values of the variables after one iteration of the C and D functions.

S.No.	M	h_0	h_1	$H(M)$
1	0x80e19efb	0x9feaad6c	0x58b49a48	11a208c1 822c7b31 c41dd0a4 10a9c8c0
	0x8e474d73			
2	0x1f399d6d	0xa148201c	0x97094b03	dd7d4e3a b426513a 6631c011 9241384f
	0x90adacf0			

5 Conclusions and Open problems

In this paper we presented collisions for the hash function Tav-128. The collisions presented are for 32-bit messages but are easily extended to messages of any length by appending any randomly chosen message to these colliding message pairs. We can state that Tav-128 is not a cryptographically secure hash function. However, the design of Tav-128 provides the starting point of a useful primitive which can have potential applications in future light weight protocols.

The construction of a library of light weight primitives having collision resistance and difficult inversion property is an open problem. Such a library will certainly be of use to designers of light weight secure protocols, not limited to just RFID applications. The cost comparison of such

primitives and conditions on the optimal cost of an individual primitive also remain interesting open problems.

References

1. Paolo Bertasi, Marco Bressan, and Enoch Peserico. *psort*, Yet Another Fast Stable Sorting Software. In *Experimental Algorithms*, volume 5526 of *Lecture Notes in Computer Science*, pages 76–78. Springer, 2009.
2. Eun Young Choi, Su-Mi Lee, and Dong Hoon Lee. Efficient RFID Authentication Protocol for Ubiquitous Computing Environment. In Tomoya Enokido, Lu Yan, Bin Xiao, Daeyoung Kim, Yuan-Shun Dai, and Laurence Tianruo Yang, editors, *EUC Workshops*, volume 3823 of *Lecture Notes in Computer Science*, pages 945–954. Springer, 2005.
3. T. Dimitriou. A Lightweight RFID Protocol to protect against Traceability and Cloning attacks. In *First International Conference on Security and Privacy for Emerging Areas in Communications Networks (SecureComm 2005), Athens, Greece*, pages 56–66. IEEE Computer Society, September 2005.
4. Martin Feldhofer, Sandra Dominikus, and Johannes Wolkerstorfer. Strong Authentication for RFID Systems Using the AES Algorithm. In Marc Joye and Jean-Jacques Quisquater, editors, *CHES*, volume 3156 of *Lecture Notes in Computer Science*, pages 357–370. Springer, 2004.
5. Martin Feldhofer and Christian Rechberger. A Case Against Currently Used Hash Functions in RFID Protocols. In Robert Meersman, Zahir Tari, and Pilar Herrero, editors, *OTM Workshops (1)*, volume 4277 of *Lecture Notes in Computer Science*, pages 372–381. Springer, 2006.
6. Dirk Henrici and Paul Müller. Hash-based Enhancement of Location Privacy for Radio-Frequency Identification Devices using Varying Identifiers. In *PerCom Workshops*, pages 149–153. IEEE Computer Society, 2004.
7. Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997. Available at `http://www.cacr.math.waterloo.ca/hac/`.
8. Pedro Peris-Lopez, Julio César Hernández Castro, Juan M. Estévez-Tapiador, and Arturo Ribagorda. An Efficient Authentication Protocol for RFID Systems Resistant to Active Attacks. In Mieso K. Denko, Chi-Sheng Shih, Kuan-Ching Li, Shiao-Li Tsao, Qing-An Zeng, Soo-Hyun Park, Young-Bae Ko, Shih-Hao Hung, and Jong Hyuk Park, editors, *EUC Workshops*, volume 4809 of *Lecture Notes in Computer Science*, pages 781–794. Springer, 2007.
9. Keunwoo Rhee, Jin Kwak, Seungjoo Kim, and Dongho Won. Challenge-Response Based RFID Authentication Protocol for Distributed Database Environment. In Dieter Hutter and Markus Ullmann, editors, *SPC*, volume 3450 of *Lecture Notes in Computer Science*, pages 70–84. Springer, 2005.
10. Stephen A. Weis, Sanjay E. Sarma, Ronald L. Rivest, and Daniel W. Engels. Security and Privacy Aspects of Low-Cost Radio Frequency Identification Systems. In Dieter Hutter, Günter Müller, Werner Stephan, and Markus Ullmann, editors, *SPC*, volume 2802 of *Lecture Notes in Computer Science*, pages 201–212. Springer, 2003.

A Some properties of functions A and B

In this section we comment on some combinatorial properties of the functions A and B. As already mentioned, these functions are not permutations over input messages.

A.1 Analyzing function A

The `for` loop runs 32 times and updates h_0. Let the initial value of h_0 before the loop starts be $h_{0,0}$ and the updated value of h_0 after the i^{th} iteration be $h_{0,i}$. In step i, the loop will perform the following operation.

$$h_{0,i} = (h_{0,i-1} \ll 1) + (h_{0,i-1} + m) \gg 1.$$

We therefore analyze the following equation.

$$x = (y \ll 1) + (y + m) \gg 1 \tag{1}$$

In the equation above, there are 3 variables: x, y and m. Consider the problem of solving for y given x and m. We make the following observations on this problem.

1. For all pairs of x and m, there exist two distinct values of y satisfying Equation 1.
2. Let these two values of y be y_1 and y_2. The difference of y_1 and y_2 is always `0x55555555` (or its additive inverse modulo 2^{32}, *i.e.*`0xaaaaaaab` since the addition in Equation 1 is modulo 2^{32}). Note that the 32-bit constant `0x55555555` represents alternating sequence of 0 and 1 bits.
3. Note that Equation 1 can be written as

   $$(x - y \ll 1) = (\text{something}) \gg 1.$$

   The most significant bit (msb) of the rhs is always zero, hence the msb of $(x - y \ll 1)$ must also be always 0.

Despite the inversion of a single step of Equation 1 being trivial for a message m (given x and y), the problem of inverting the `for` loop (i.e. finding an m given $h_{0,0}$ and $h_{0,32}$) is difficult.

A.2 Analyzing function B

Similar to the analysis above, let us try to study the following equation.

$$(y \ll 1) + (y \gg 1) + y + m = x \tag{2}$$

We would like to solve Equation 2 for y, given the values of x and m. It is interesting that unlike in the case of Equation 1, this time we may or may not be able to solve the equation. The following cases can occur:

1. For some values of (x, m), there is no solution for y. For example: $x =$ `0x7409c642` and $m =$ `0x3d303017`.
2. For some values of (x, m), there is exactly one value of y satisfying Equation 2. E.g. $x =$ `0x152e1fdb` and $m =$ `0x3d77b373`, for which $y =$ `0xcfeafa67`.
3. For some values of (x, m), there are two values of y satisfying Equation 2. E.g. $x =$ `0x4a6cd8f5` and $m =$ `0x10dff39b`, for which $y =$ `0x5995f863` and `0xa2ba8aac`.

The probability of occurrence of the three cases above are roughly $\frac{1}{3}$.

Similar to the case of function A, the inversion of the `for` loop (computing m from $h_{1,0}$ and $h_{1,32}$ where the symbols have similar meaning) corresponding to 32 calls to Equation 2 is a difficult problem.

B Tav-128 reference code from [8]

```
/*****************************************************************/
Process the input a1 modifying the accumulated hash a0 and the state
/*****************************************************************/
void tav(unsigned long *state, unsigned long *a0, unsigned long *a1)
{
unsigned long h0,h1;
int i,j,r1,r2,nstate;

/* Initialization */
r1=32; r2=8; nstate=4;
h0=*a0; h1=*a0;

/* A - Function */
for(i=0;i<r1;i++){h0=(h0<<1)+((h0+(*a1))>>1);}
/* B - Function */
for(i=0;i<r1;i++){h1=(h1>>1)+(h1<<1)+h1+(*a1);}

/ * C and D - Function */
for(j=0;j<nstate;j++) {
   for(i=0;i<r2;i++) {
         /* C - Function */
            h0^=(h1+h0)>>3;
            h0=((((h0>>2)+h0)>>2)+(h0<<3) +(h0<<1))^0x736B83DC;
         /* D - Function */
             h1^=(h1^h0)>>1;
             h1=(h1>>4)+(h1>>3)+(h1<<3)+h1;
   } // round-r2
   state[j]+=h0;
   state[j]^=h1;
} // state
```

```
/* a0 updating */
 *a0=h1+h0;
}

/*****************************************************************************/
Initialization of the state and a0 with random values obtained from www.random.org
/*****************************************************************************/
void init state(unsigned long *state, unsigned long *a0)
{
   state[0]=0xa92be51d;
   state[1]=0xba9b1ef0;
   state[2]=0xc234d75a;
   state[3]=0x845c2e03;
   a0[0]=0x768c7e74;
}
```