CS641A: Assignment 6

Rijndael
Abhishek Jain, Arham Chopra, Ayush Tulsyan
13027, 14130, 14807167

Inverting χ and θ

Inverting χ

χ is defined as:
Input:
state array A.
Output:
state array A'.
Steps:

1. For all triples (x, y, z) such that $0 \leq x \leq 4, 0 \leq y \leq 4$, and $0 \leq z \leq 7$, let $A'[x, y, z] = A[x, y, z] \oplus ((A[(x + 1)mod5, y, z] \oplus 1).A[(x + 2)mod5, y, z])$. (1)
2. Return A'.

We tried transforming all possible inputs from $0 - 31$ using χ using our code(breaking_chi.py). We found that there is a unique output for each input, thus this transformation is invertible.

From (1) it is clear that if we know two consecutive elements $A[(x+1)mod5, y, z], A[(x+2)mod5, y, z]$, then we can find all of the other $A[*, y, z]$. This is because each $A[x, y, z]$ only depends on its two successors. By knowing any two consecutive elements we can iteratively find all the predecessors.

$$A[x, y, z] = A'[x, y, z] \oplus ((A[(x + 1)mod5, y, z] \oplus 1).A[(x + 2)mod5, y, z]).$$

Next we consider two cases

- $A[x + 1, y, z] = 1$, then $A[x, y, z] = A'[x, y, z]$ again from (1). Thus if we know some $A[x, y, z]$ to be 1 then we will also know of two consecutive elements and we will be able to find all of $A[*, y, z]$.
- $A[x + 2, y, z] = 0$, then $A[x, y, z] = A'[x, y, z]$ again from (1). If $A[x, y, z] = 1$ then form previous point, we can find all the elements. Otherwise $A[x, y, z] = 0$, then we repeat the current point and we obtain $A[x - 2, y, z]$. We keep repeating this till we get a 1 or we have found all elements already. This will terminate since the no of elements in the $A[*, y, z]$ is odd, and we are jumping with steps of 2(even)(and 2,5 are coprime). If we never get a 1, then we will find all elements to be 0.

From above it is clear that if we have knowledge of one element in a row, then know the complete row. Thus we can very easily invert any row by trying $\{0, 1\}$ for and element of the row and verifying that (1) is satisfied for all the elements of the row.

Now to invert the whole block we have to repeat the above procedure for each row of the block. Since there are 40 rows in the block(8*5), and there are 2 possibilities for each row, we have a total of 80 possibilities. This is because each row transformation is independent of the other. Thus we can invert χ easily by trying out all these 80 possibilities.

Inverting θ

θ is defined as:
Input:
state array A.
Output:
state array A'.
Steps:

1. For all triples (x, z) such that $0 \leq x \leq 4$, and $0 \leq z \leq 7$, let
   $C[x, z] = A[x, 0, z] \oplus A[x, 1, z] \oplus A[x, 2, z] \oplus A[x, 3, z] \oplus A[x, 4, z]$ (1)

2. For all triples (x, z) such that $0 \leq x \leq 4$, and $0 \leq z \leq 7$, let
   $D[x, z] = C[(x-1) mod 5, z] \oplus C[(x+1) mod 5, (z-1) mod 8]$ (2)

3. For all triples (x, y, z) such that $0 \leq x \leq 4, 0 \leq y \leq 4$, and $0 \leq z \leq 7$, let
   $A'[x, y, z] = A[x, y, z] \oplus D[x, z]$ (3)

Consider any $A[x, y, z], A[x, y', z]$, from (3) it is clear that

$$A'[x, y, z] \oplus A'[x, y', z] = A[x, y, z] \oplus A[x, y', z]$$

And thus if we know one of $A[x, *, z]$ then we can find all the others of the same column, using

$$A[x, y, z] = A[x, y, z] \oplus A'[x, y, z] \oplus A'[x, y', z]$$

Also $(1), (2)$ can be written in terms of only one element of each column (x, z) as

$$C[x, z] = A[x, 0, z] \oplus A'[x, 1, z] \oplus A'[x, 2, z] \oplus A'[x, 3, z] \oplus A'[x, 4, z]$$
$$D[x, z] = A[(x-1) mod 5, 0, z] \oplus A[(x+1) mod 5, , 0, (z-1) mod 8] \oplus \text{some A' terms}$$

Thus (3) it can be shown that $A[x, y, z]$ only depends on two successive elements of an adjacent row within the same plane.

$$A'[x, y, z] = A[x, y, z] \oplus A[x-1, 0, z] \oplus A[(x+1, , 0, z-1] \oplus \text{some A' terms}$$
$$A[x+1, 0, z-1] = A[x-1, 0, z] \oplus A[x, y, z] \oplus \text{some A' terms} \quad (4)$$

Thus given one row in a plane $A[*, y, z]$, all other rows can be found $A[*, y, *]$. If all elements in a plane are found then all other can be found as well, since knowledge of one element in a column can be used to get all others as well in the same column.

By assuming one row, say the $A[*, 0, 7]$ to be $(a_0, a_1, a_2, a_3, a_4)$, we can get all other elements in the plane in terms of these a_i and some $A'[x, y, z]$ terms. Thus we can get all elements of the row $A[*, 0, 0]$ as $(a'_0, a'_1, a'_2, a'_3, a'_4)$ in terms of $(a_0, a_1, a_2, a_3, a_4)$ as well. Since (4) is valid for all elements, it can be applied to get $(a_0, a_1, a_2, a_3, a_4)$ in terms of $(a'_0, a'_1, a'_2, a'_3, a'_4)$.

Thus we can obtain 5 independent equations for the 5 variables of we started off with. This equations can be solved to get the solution for $(a_0, a_1, a_2, a_3, a_4)$. Now that a_i are found, everything else can also be found. One way to solve the equations would be by cross verification. Thus we try all possible 0-31 inputs for a row and check if all the conditions verify(this can be seen in the file breaking_theta.py). Thus we are able to invert θ.

Security of WECCAK for $H = R \circ R$, or $H = R \circ R$ .. *24 times* .. $R \circ R$

Collision Attack

Consider any message m, running the transformations R over m twice gives a block,B, with some value at the last 16 elements of the B. This is attack is independent of what H is.

If somehow it can be ensured that two messages m_1, m_2 (each 184 bits) have the same last 16 bits, then we can generate two messages m_1', m_2', that will collide. Let the starting 184 bits of m_1, m_2 be u_1, u_2. We can generate a pair of messages h_1, h_2 each 184 bits, such that

$$u_1 \oplus h_1 = u_2 \oplus h_2$$

This can be easily done by equating both to any random 184 bits string and evaluating h_1, h_2 appropriately.

Now consider the messages $m_1' = m_1, h_1, m_2' = m_2, h_2$. After xoring with h_1, h_2, the blocks will become identical and thus will give that same output block and hash.

By passing atmost $2^{16} + 1$ messages, we are bound to get a collison among the last 16 bits for atleast one pair of messages, by the Pigenhole Principle. By birthday paradox, on passing approx 2^8 messages, we can get a collision among the last 16 bits atleast once with a high probability.

Second Pre-image attack

Consider any message m, and its hash be $H(m)$. This is attack is independent of what H is.

We have to find a second message m_2, such that its hash is also $H(m)$. Also let the round function be R, thus the output block for message m will be $R(m)$ and $H(m)$ will just be the first 80 bits of $R(m)$.

If somehow a message can be found such that the last 16 bits of the hash are 0, then we can find a 2nd preimage for any message. Let this message be $m*$. Consider the message $m_2 = m*, m'$, where the following holds

$$\begin{aligned}
R(m*)[184:199] &= \text{"00...000"} \\
m' &= R(m*)[0:183] \oplus m \\
R(m*)[0:183] \oplus m' &= m \\
H(m*, m') &= R((R(m*)[0:183] \oplus m')||R(m*)[184:199])[0:79] \\
H(m*, m') &= R(m||\text{"00..00"})[0:79] \\
H(m*, m') &= H(m)
\end{aligned}$$

Thus if $m*$ can be found than for any m, a second pre-image can be easily found.

To find this $m*$, we tried hashing random input messages and checked if the output block has last 16 bits as 0. We wrote a code for this that can be found in utilities.py . We basically ran a brute force code to get this $m*$, and we were able to get it in around 100,000 iterations most of the times.

The message we got in bitstring for $H = R \circ R$ is

$$\begin{aligned}
m* = &\,0001011010101110110101000100110010000010101111 \\
&||1001110100101000001101011100101100100000100011 \\
&||0110010111110011001000000000111000011001001000 \\
&||0011000000001010100110011101011000100100000100
\end{aligned}$$

The message we got in bitstring for $H = R \circ R$ .. *24 times* .. $R \circ R$ is

$$\begin{aligned} m* = & 00011011110111011110001011100111001111011000 11 \\ & ||11010001011110001101111000110010111101111011 01 \\ & ||01011100100101100110001001111011011110000000 11 \\ & ||00101010010011110011000000001100001010000000 11 \end{aligned}$$

Pre-image attack

Since the constituents of F are invertible, the F, whether H_1 or H_2 is also invertible. So, given any digest of 80 bits, one can append it with random bits and invert to get an input.

This input can't be coined as a pre-image, since this may or may not have the last 16 bits as zero. To get around this restriction, we formulate a Meet in the Middle Attack.

Consider a message of length between 185 and (184*2)=368 characters(inclusive). This message consists of two blocks. The first block will be used to generate the last 16 bits and the second block will help us modify the digest of first block to get the required input for second block.

We take random bit-strings input for first block and look at the digest obtained for the first block. We create a set of these random inputs which map to a significant subset of all possible values of the last 16 bits. The size of this subset will be decided later. Now, we do the same from the other end. We consider the 80 bit digest and append 120 bit random strings to it. Passing it through F^{-1}, we get the input to second round of F. The second block of message is XORed with the first 184 bits, so, we could obtain any of the required input for this 184 bits. But the last 16 bits can't be modified. These 16 bits are being carried over from the previous block. We iterate over random 120 bits strings to get a set of inputs(to second round of F) which map to the 80 bit digest. The size of this set will also be decided soon.

Currently, we have a set of inputs to first round of F and their corresponding output values for the last 16 bits. We also have a set of inputs to second round of F and we know that these map to the required digest. We will have a pre-image if we could find a element in the first set, whose output from F will map to some 16 bit suffix which is also present in the other set. This collision occurs with probability 50% if the size of both these sets is nearly 2^8.

Otherwise, we have iterated over the inputs to the first round of F to generate all 2^{16} suffixes. If one could assume that if the input to F is random, then all outputs bits are also random, then the total number of random inputs required to generate all 2^{16} suffixes is $2^{16} \ln 2^{16}$ (This many iterations will generate all suffixes with high probability).

The simulation of above took 737803 iterations of random bit strings to generate all 2^{16} suffixes for H_1. The code for this simulation is present in `populate.py`. The value $2^{16} \ln 2^{16}$ is approximately 729042, which is pretty close to our number of iterations.

References

[1] SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, FIPS 202, 08/2015