On the Practical Implementation of Impossible Differential Cryptanalysis on Reduced-Round AES

Sourya Kakarla(✉), Srinath Mandava, Dhiman Saha, and Dipanwita Roy Chowdhury

Crypto Research Lab, Department of Computer Science and Engineering, IIT Kharagpur, Kharagpur, India
{skakarla,smandava,dhimans,drc}@cse.iitkgp.ernet.in

Abstract. In this work, we give a practical implementation of the well known impossible differential attack on 5 round AES-128 given by Biham and Keller. The complexity of the original attack is in the order of the practical realm with time complexity 2^{31} and data complexity $2^{29.5}$. However, the primary memory required to execute the attack was 4 TB making it difficult to implement which is supported by the fact that there are no reported implementations of the attack. We propose a data-memory tradeoff for the attack which lets us reduce memory needed at the expense of increased data complexity. We have been able to implement the attack using 128.5 GB of primary memory and 2^{32} data complexity. Though the data complexity is increased by about 4.65 times, it makes up for the fact that we decreased the memory usage by about 32 times. We also extend the implementation to 5 round AES-192/256. To the best of our knowledge, the implementations of attacks in this work are the first ones available publicly.

Keywords: AES · Impossible differential · Cryptanalysis · Data structure · Implementation · Data-memory tradeoff · Key recovery

1 Introduction

AES [6] has become the standard symmetric key cipher used across the internet since the Rijndael family of ciphers were selected for Advanced Encryption Standard by the U.S National Institute of Standards and Technology. Its security is of utmost importance to the security of the internet thereby making it one of the most widely studied ciphers. Though there are no practical attacks possible on full round AES, round-reduced versions were attacked with complexities significantly less than the brute force complexity. These attacks help in developing greater insight into the security of AES. One such family of attacks is impossible differential attacks [2]. Impossible differential cryptanalysis is a form of differential cryptanalysis [4]. While traditionally differential cryptanalysis tracks the probabilities of differences through the rounds of a cipher that are

L. Batten et al. (Eds.): ATIS 2017, CCIS 719, pp. 58–72, 2017.
DOI: 10.1007/978-981-10-5421-1_6

highly probable, impossible differential cryptanalysis tracks difference patterns that are impossible to occur at an intermediate state i.e. their probability is zero. Generally, two plaintexts are taken and their difference is tracked down in the encryption direction to an intermediate state. Their corresponding ciphertexts are taken and their differences are tracked in the direction of the decryption to the same intermediate state. If at this intermediate state, the probability of both the paths holding is 0, then these patterns are called *impossible differential paths*. Using impossible differential paths, we can mount an attack by eliminating all keys which give the impossible differential path thereby leaving behind the right key.

The first impossible differential attack on `AES` is an attack on 5 round `AES-128` by Biham and Keller [3]. This was extended to a 6 round attack by Cheon *et al.* [5]. Later in 2004, Phan [10] extended the attack to 7 round `AES-192` and `AES-256`. There have been several other impossible differential attacks [1,8,9] on the different variants of `AES` since then. Among these attacks, the ones [3,5] on `AES-128` which do not exploit the key schedule of `AES` are extendable to `AES-192` and `AES-256`.

Almost all of the impossible differential attacks on `AES` with the exception of [3] are not practical attacks since their time and/or data complexities are highly infeasible to be implemented in real life. As the 5 round attack on `AES-128` is practical with time complexity only 2^{31}, we were surprised to observe that there are no readily available public implementations of the attack. We deduced that the memory needed for the attack which was 4 TB might have been the reason for it not having been implemented. So we set out to achieve probably the first implementation of a full key recovery impossible differential attack on 5 round `AES`.

Our contribution is as follows:

- First, we propose a data-memory tradeoff to the attack given by Biham and Keller [3] on 5 round `AES-128`. This tradeoff enables us to implement the attack with flexible resources.
- We implement full key recovery attack on 5 round `AES-128` using the data-memory tradeoff. This implementation uses 128.5 GB of primary memory and 2^{34} chosen plaintexts.
- We used customized data structures which lead to a time and space efficient implementation.
- Using the techniques above, we implement the attack on 5 round variants of `AES-192` and `AES-256`.

The rest of the paper is organized as follows. Section 2.1 provides a brief description of `AES` with its operations. In Sect. 2.2, notations for some of the aspects of `AES` are furnished. Section 2.4 gives an explanation of the attack given in [3]. The data-memory tradeoff for the attack is given in Sect. 3. The implementation details of the attacks on `AES-128`, `AES-192`, and `AES-256` are given in Sect. 4. Finally, we finish with concluding remarks in Sect. 5.

2 Background and Preliminaries

2.1 AES

AES is a block cipher which operates on blocks of 128 bits using key sizes of 128, 192 or 256 bits. The intermediate state of AES can be arranged as 4×4 matrix as shown in Fig. 1 The number of rounds varies with the key size, 10 for AES-128, 12 for AES-192 and 14 for AES-256. Each round of AES encryption consists of the following operations in the given order:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Fig. 1. Byte positions in AES state

- SubBytes(SB): Each byte is substituted by the value from S-box. This is the only non-linear operation in AES.
- ShiftRows(SR): Each row in the state is shifted cyclically to the left. The i^{th} row is shifted by i bytes to the left ($0 \leq i \leq 3$).
- MixColumns(MC): Each column is multiplied by a constant 4×4 matrix over the $GF(2^8)$.
- AddRoundKey(ARK): The state is XORed with the 128-bit round key.

The decryption of AES consists of the inverse of the above operations in the reverse order. Last round of AES does not contain a MixColumns operation. It is assumed that this is true for round reduced AES too. An additional AddRoundKey is applied at the start of the first round. Thus for n rounds, there are $n+1$ round keys. The round keys are supplied by the key schedule algorithm.

2.2 Notation

We define notations to represent intermediate states and other aspects of the AES algorithm. There are n rounds and $1 \leq i \leq n$. The key is represented by K.

- s_i is the state at the beginning of i^{th} round.
- s_0 is the initial state which is generally the plaintext.
- s_{n+1} is the final state which is generally the ciphertext.
- s_1 is the state after the initial AddRoundKey.
- $s_i^{\mathcal{B}}$ is the state after SubBytes of i^{th} round.
- $s_i^{\mathcal{R}}$ is the state after ShiftRows of i^{th} round.
- $s_i^{\mathcal{M}}$ is the state after MixColumns of i^{th} round ($i \neq n$).
- $s_i^{\mathcal{A}}$ is the state after AddRoundKey of i^{th} round.

- $s_{i,j}^{\mathcal{X}}$ is the j^{th} byte of $s_i^{\mathcal{X}}$ $(1 \leq j \leq 16,\ \mathcal{X} \in \{\varnothing, \mathcal{B}, \mathcal{R}, \mathcal{M}, \mathcal{A}\})$.
- K_i denotes the round key used in round i. K_0 corresponds to the key used in the initial `AddRoundKey`.
- $\mathcal{D}_p$: The set of byte positions in $s_1^{\mathcal{B}}$ which move to the same column after `ShiftRows` in $s_1^{\mathcal{R}}$.

$$\mathcal{D}_p = ((1, 6, 11, 16), (2, 7, 12, 13), (3, 8, 9, 14), (4, 5, 10, 15))$$

- $\mathcal{D}_c$: The set of byte positions in $s_n^{\mathcal{A}}$ which move to the same column after `ShiftRows`$^{-1}$ in $s_n^{\mathcal{R}}$.

$$\mathcal{D}_c = ((1, 8, 11, 14), (2, 5, 12, 15), (3, 6, 9, 16), (4, 7, 10, 13))$$

We can see that $s_i^{\mathcal{A}} = s_{i+1}$, $1 \leq i \leq n$.

2.3 4 Round Impossible Differential

In this subsection, the 5 round attack on `AES-128` given by Biham and Keller [3] is explained. A 4 round impossible differential path which is used in the attack is shown in Fig. 2. The difference between the two states are traced through the rounds of `AES`. The last round does not contain the `MixColumns` operation. The grey squares represent bytes where the difference is non-zero. These bytes are called *active bytes*. The white squares represent bytes where the difference is zero i.e. the bytes where the states are equal. These bytes are called *passive bytes*.

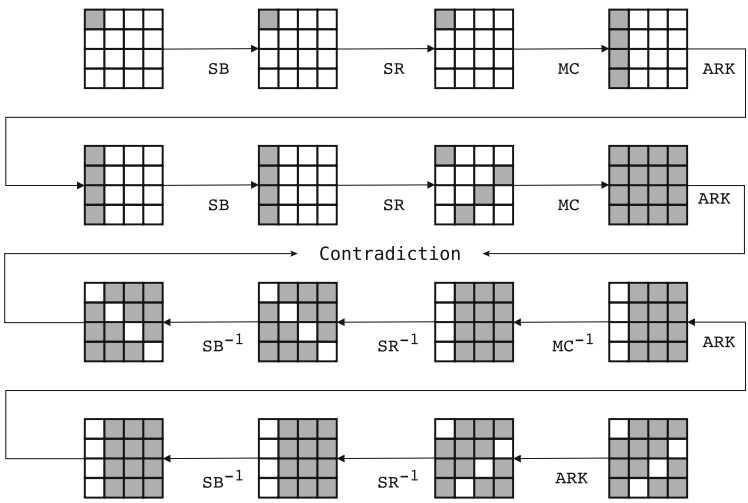


Fig. 2. 4 round impossible differential path

If the two states are p and q, the difference between the states is represented as another state, d, where $d_i^{\mathcal{X}} = p_i^{\mathcal{X}} \oplus q_i^{\mathcal{X}}, \forall \mathcal{X} \in \{\varnothing, \mathcal{B}, \mathcal{R}, \mathcal{M}, \mathcal{A}\}$. The path shows

that if there is a single active byte in d_0, all the bytes are active in $d_2^{\mathcal{A}}$. If all the bytes along one of the diagonals in $\mathcal{D}_c$ are passive in $d_4^{\mathcal{A}}$ (ciphertext pair), then at least 4 bytes are passive in d_3. This is a contradiction since $d_2^{\mathcal{A}} = d_3$. This proves that if the starting states differ only in one byte, then it is impossible to obtain ciphertext states that are passive along one of the diagonals in $\mathcal{D}_c$.

2.4 5 Round Attack on AES-128 by Biham and Keller

A round is added at the top to the 4-round impossible differential path to mount an attack on 5 round AES-128 as shown in Fig. 3. All keys which give the impossible condition in the last 4 rounds can be deemed as wrong keys and be eliminated.

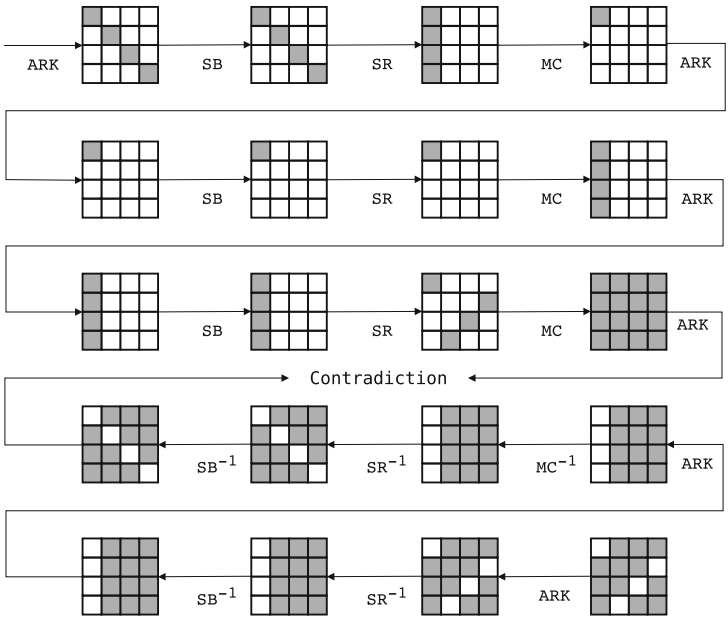


Fig. 3. 5 round attack on AES-128

Let the plaintext pairs which are active in one of the diagonals from $\mathcal{D}_p$ be referred to as *chosen pairs* and the chosen pairs whose ciphertext pairs are passive in one of the diagonals from $\mathcal{D}_c$ be referred to as *desired pairs*. For each desired pair we can eliminate all key guesses of K_0 which give a single active byte in $d_1^{\mathcal{M}}$. If we fix the diagonal (from $\mathcal{D}_p$) in plaintext pairs, the property is affected by only the key bytes along that diagonal. These set of bytes are called as a *partial key* corresponding to that diagonal. Thus a partial key can be independently eliminated irrespective of the other partial keys of K_0.

Instead of taking a desired pair and checking whether the property satisfies for all the 2^{32} partial key guesses, pre-computation is used to speedup the elimination of wrong partial keys. A hash table is created using Algorithm 1 which

Algorithm 1. Compute Hash Table

```
HashTable← InitialiZation()
for all a, a', b, c, d ∈ {0, 1, ··· , 2^8 − 1}, a ≠ a' do        ▷ These are bytes.
    col ← InitializeColumn(a, b, c, d)
    col' ← InitializeColumn(a', b, c, d)
    col ← MixColumn^-1(SubBytes^-1(col))    ▷ The column version of the operations are used
    col' ← MixColumn^-1(SubBytes^-1(col'))
    hkey ← col ⊕ col'
    HashTable[hkey].Append(col)
end for
```

gives the partial keys eliminated by a desired pair. The hash table contains 2^{40} values stored in 2^{32} indexes averaging 2^8 values per index.

Given a desired pair with plaintexts P_1, P_2, we can compute all the partial keys eliminated by calculating $x \oplus P_1$ over all the values x in hash table at the index $P_1 \oplus P_2$. On an average, 2^8 partial keys are eliminated per desired pair. The probability that a chosen pair is a desired pair is $2^{-32} \cdot 4 = 2^{-30}$. We take chosen plaintexts which are varied over the 2^{32} possibilities along a diagonal from $\mathcal{D}_p$ with all the other bytes fixed. From these chosen plaintexts, we get 2^{63} chosen pairs. The corresponding ciphertexts are obtained and desired pairs are taken from the chosen pairs. We get $2^{63} \cdot 2^{-30} = 2^{33}$ desired pairs. We iterate over these desired pairs and eliminate the wrong partial keys using the hash table. The expected number of wrong partial keys remaining is

$$2^{32} \cdot (1 - 2^{-32})^{2^{33} \cdot 2^8} \approx 0 \tag{1}$$

It is given in the original attack that using 2^{28} desired pairs would suffice for the attack which would mean using about $2^{29.5}$ chosen plaintexts. Iterating through the 2^{28} pairs, we make 2^8 XOR and hash table lookups per pair making it a total of 2^{36} XOR calculations and hash table computation. If the computation cost of XOR calculation and hash table lookup is equated to 2^{-5} part of computation of 1 encryption, the total time complexity is about 2^{31} encryptions. Similarly, the precomputation of hash table is calculated as 2^{36} encryptions. The data complexity is $2^{29.5}$ chosen plaintexts. The memory used is at least 4 TB as 2^{42} bytes are used for the hash table. Although this memory can be deemed as practical, a server with 4 TB of primary memory is not yet commonly available. Thus the memory usage is a bottleneck for implementing the attack. We introduce measures in the following section to tackle this.

3 Memory Reduction Techniques

In this section, we introduce methods to reduce the primary memory needed for the attack. First, we give a simple 50% reduction in the memory needed with no extra cost and then introduce a data-memory tradeoff.

Reducing the hash table size to 2 TB
We first look to reduce the size of the hash table without any other increased costs. In the original attack both the ordered pairs $((a, b, c, d), (a', b, c, d))$ and $((a', b, c, d), (a, b, c, d))$ are used in the computation of the hash table. We can remove this redundancy by replacing the condition $a \neq a'$ with $a > a'$ in line 2 of Algorithm 1. The hash table size is reduced by half to 2 TB. Given a desired pair, (P_1, P_2) we eliminate the partial keys $x \oplus P_2$ and $x \oplus P_1$ (as opposed to only $x \oplus P_1$ in the original attack) over the values x at the index $P_1 \oplus P_2$. This makes up for the fewer values in the hash table. Even though now we have only an average of 2^7 values per index in the hash table, still on an average 2^8 partial keys are eliminated per desired pair. Note that the time and data complexities are not changed with this alteration.

Data-Memory Tradeoff
To further reduce the hash table, we need to look at the equation which calculates number of remaining partial keys. If we have an average of 2^x $(0 \leq x \leq 7)$ values per index in the hash table, 2^{x+1} partial keys are expected to be eliminated per desired pair. If 2^y desired pairs are used, the expected number of remaining partial keys is given by

$$2^{32} \cdot (1 - 2^{-32})^{2^y \cdot 2^{x+1}} = 2^{32} \cdot (1 - 2^{-32})^{2^{x+y+1}} \tag{2}$$

In the original attack with 2^8 values per index, 2^{33} desired pairs sufficed. Thus we have,

$$x + y + 1 = 8 + 28 \Rightarrow y = 35 - x. \tag{3}$$

$$2^{32} \cdot (1 - 2^{-32})^{2^{35-x} \cdot 2^{x+1}} \approx 483 \tag{4}$$

If we have 2^x values per index in the hash table, then the number of desired pairs required for the attack would be 2^{35-x}. For 2^y desired pairs used, $2^{(y+31)/2}$ chosen plaintexts are needed. We have to be careful when computing a hash table (smaller than the original) such that the values are not extremely skewed across the indices. If we have a skewed hash table, it might be possible that all the desired pairs used might get values from the sparse part of the table, thus resulting in fewer keys eliminated than expected. In practice it is found that randomly sampling a subset of all values of (a, a', b, c, d) when calculating the hash table leads to a distribution which is not too skewed.

Thus we can reduce the memory usage by reducing x to a suitable value and increasing y to the corresponding value. It is preferable not to reduce x to a value as low as 0. Even though there is 1 entry per index on an average for $x = 0$, a skewed hash table might lead to 0 entries at some indexes. If we take $x = 2$, $y = 35 - 2 = 33$. Number of chosen plaintexts required is 2^{32}. The memory needed for the hash table is 64 GB (for storing 2^{34} 4-byte values). Though the data complexity is increased by $2^{2.5}$ times, it makes up for the fact that primary memory usage is greatly reduced from 4 TB to 64 GB provided optimal implementation of the hash table.

Reducing data complexity. There might be scenarios where the attacker is equipped with abundant memory but has less data to attack with. From the data-memory tradeoff (Eq. 3), we can see that decreasing the data complexity is possible if the average number of entries per index in the hash table increases. This is actually possible if we alter the computation of the hash table. Instead of only considering pairs such as $((a, b, c, d), (a', b, c, d))$, we can also consider the pairs $((a, b, c, d), (a, b', c, d)), b \neq b'$ and similar pairs with the different byte at c or d. Now the maximum of average number of entries per index in the hash table is increased from 2^7 to 2^9.

Thus either the memory usage or the data complexity can be decreased based on the requirements. The memory used and the data complexity for different x is given in Table 1.

Table 1. Attack on 5 round `AES-128` data-memory tradeoff (partial key)

Average number of hash table entries per index x	Hash table entries 2^{32+x}	Hash table size 2^{x+4} GB	Desired pairs 2^{35-x}	Chosen plaintexts $2^{33-0.5x}$
9	2^{41}	8 TB	2^{26}	$2^{28.5}$
7	2^{39}	2 TB	2^{28}	$2^{29.5}$
4	2^{36}	256 GB	2^{31}	2^{31}
3	2^{35}	128 GB	2^{32}	$2^{31.5}$
2	2^{34}	64 GB	2^{33}	2^{32}
1	2^{33}	32 GB	2^{34}	$2^{32.5}$

Time Complexity. One might expect that with increase in data complexity, time complexity might increase. This is in fact not true. The data-memory tradeoff has no effect on the time-complexity of the attack. The main operation in the attack is calculating the partial keys eliminated for each desired pair from the hash table. Thus time complexity is proportional to $2^x \cdot 2^y = 2^{x+y}$, which is constant as per Eq. 3. So we expect that time complexity to not be affected by changes in x.

4 Implementation Details and Experimental Results

The implementation of the attacks was coded in `C`. The code was run on a server with `Intel(R) Xeon(R) CPU E5-2697 v2 @ 2.70GHz` processors. The total primary memory of the server was 256 GB under a NUMA setting. An `AES` implementation using `AES-NI` [7] instructions was used. Other subroutines of the attack which involved `AES` operations also used these instructions. Now we give some high level implementation details of the attack.

4.1 Implementation for AES-128

From the data-memory tradeoff, we take $x = 2$, meaning the hash table has an average of 2^2 entries per index. The number of chosen plaintexts used is 2^{32} per partial key.

Hash Table. For any hash table implementation, we expect the size to be at least 64 GB. But as the number of entries per index vary, it is difficult to implement the table without using any extra memory. We implemented a data structure for the hash table as shown in Fig. 4.

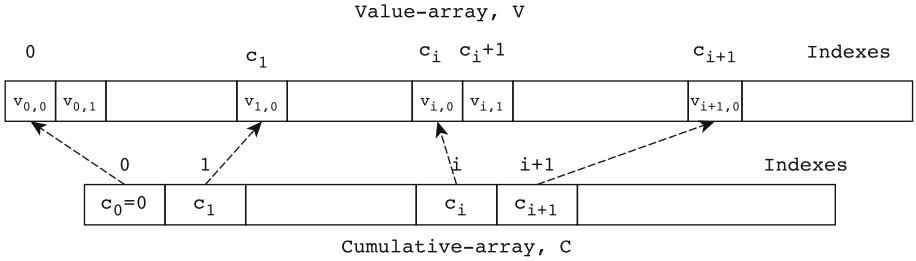


Fig. 4. The hash table data structure with the arrays V and C. $v_{i,j}$ is the j^{th} value in the index i.

If n_i is the number of values at index i, the cumulative count c_i is given by

$$c_i = \begin{cases} 0, & \text{for } i = 0 \\ \sum_{k=0}^{i-1} n_k, & \text{for } i > 0 \end{cases}$$

V is the *value-array* where all the values of the hash table are stored in the order of their hash table indexes. C is the *cumulative-array* where $C[i] = c_i$. The hash table values at index i are given by

$$\texttt{HashTable}[i] = \Big\{ V[j] \mid C[i] \leq j < C[i+1] \Big\}$$

Each element of V is 4 bytes in size since hash table values are 4 byte values. As there are 2^{34} hash table values, size of V is 64 GB. For elements in C, 4 bytes is not sufficient as $\exists i, C[i] \geq 2^{32}$. Therefore, we use 8 byte sized elements. There are 2^{32} elements in C making its size 32 GB. Thus, the total size of hash table is 96 GB.

Desired Pairs. We would be needing 2^{33} desired pairs from 2^{32} chosen plaintexts for a partial key. However, it is not feasible to enumerate the 2^{63} pairs of chosen plaintexts to obtain the desired pairs. This is avoided by fixing the

ciphertext diagonal (from $\mathcal{D}_c$) and storing the chosen plaintexts in a hash table called *desired-table*. This table is indexed by the value of the diagonal bytes of the corresponding ciphertexts. We only need to store the bytes of the plaintext along the plaintext diagonal as all the chosen plaintexts are equal across the other bytes. The procedure for constructing the desired-table is given in Algorithm 2. Once the desired-table is constructed, all the pairs of plaintexts at the same index are desired pairs. This process is repeated for different ciphertext diagonals to get all the desired pairs.

Algorithm 2. Desired Pair Generation 5 round `AES-128`

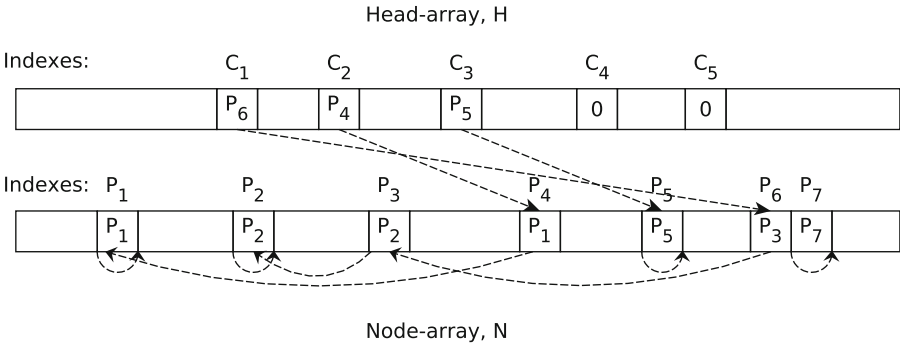
Require: $pdiag, cdiag \in \{0, 1, 2, 3\}$ ▷ indexes of the plaintext and ciphertext diagonals resp.

procedure BUILDDESIREDTABLE($pdiag, cdiag$)
  `DesiredTable`← INITIALIZATION()
  for all $a, b, c, d \in \{0, 1, \cdots, 2^8 - 1\}$ **do** ▷ These are bytes
    $p \leftarrow$ INITIALIZESTATEPDIAGONAL($(a, b, c, d), pdiag$) ▷ Bytes not in diagonal are fixed
    $c \leftarrow$ ENCRYPTIONORACLE(p)
    $dc \leftarrow$ GETCDIAGONALBYTES($c, cdiag$) ▷ Obtain bytes in ciphertext diagonal
    `DesiredTable`$[dc].Insert((a, b, c, d))$
  end for
  return `DesiredTable`
end procedure

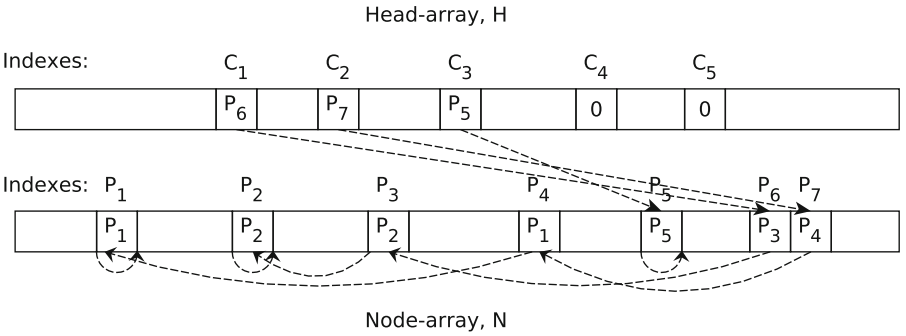
As the number of entries per index in the desired-table is not known, we use a linked-list like data structure shown in Fig. 5 which is highly space efficient. H is the *head-array*. $H[i]$ gives the first node at index i. N is the *node-array*. $N[i]$ is the next node in the list. Ending of a list is represented by a self-reference at the last node of the list. There are 2^{32} 4-byte sized elements in both the arrays. Thus, the size of each array is 16 GB making the total size of desired-table 32 GB. The procedures of desired-table are given in Algorithm 3. Some details of null references in H are skipped for brevity. Elements of H are initialized to 0 while elements of N are initialized as $N[i] = i$.

Key Elimination. Once the desired-table for a ciphertext diagonal is obtained, we eliminate the partial keys over the corresponding desired pairs using the hash table. This is repeated for all the diagonals. The overall implementation is given as a procedure in Algorithm 4. 2^{32} bits are needed to keep track of the eliminated partial keys. Therefore, 0.5 GB is used for the *key-array*. The over all memory usage is displayed in Table 2.

From Eq. 4, it is expected that around 483 partial keys to be not eliminated. In practice we have found that around 450 to 520 partial keys remain after the elimination process. It took 12 h to get these keys for one partial key. We get the



(a) $\{P_2, P_3, P_6\}$ have the same ciphertext diagonal value C_1. Similarly, $\{P_1, P_4\}$ have C_2 as their ciphertext diagonal. Only P_5 has C_3 as its ciphertext diagonal. P_7 has not been processed yet. C_4 and C_5 have no entries yet.



(b) P_7 is processed and it is found that its ciphertext diagonal is C_2. Hence, now $\{P_1, P_4, P_7\}$ are the plaintexts which have C_2 as their ciphertext diagonal.

Fig. 5. Two intermediate states of the desired-table during its construction.

correct key, $K = K_0$, by brute forcing over all the possible keys computed from the partial key sets (Table 3).

As the attack on `AES-128` does not involve key schedule, we can apply it to the 5 round `AES-192` and `AES-256` variants by making a few alterations which are explained in the following sub-section.

4.2 Implementation for `AES-192/256`

Here we explain the implementation for `AES-256` which can be applied for `AES-192` without any changes. The first round key K_0 is obtained in the same way as for `AES-128`. For the second round key K_1, we give a new differential path that shifts the previous path by one round forward as shown in Fig. 6. Note that the definition of desired pair needs to be modified as in the ciphertext pair, only one passive byte is required for it to be a desired pair for K_1. Thus data complexity for K_1 is significantly less than that for K_0. Similarly we

Algorithm 3. Desired Table Procedures

procedure INSERT(p, c) ▷ Insert p in index c
  if $H[c] = 0$ **then** ▷ List at index c is empty
    $H[c] \leftarrow p$
  else
    $N[p] \leftarrow H[c]$
    $H[c] \leftarrow p$ ▷ Inserting p at the start of the list at $H[c]$
  end if
end procedure
procedure GETENTRIES(c) ▷ Returns all entries at index c
  $set \leftarrow \varnothing$
  $next \leftarrow H[c]$
  if $next = 0$ **then** ▷ If list at index c is empty
    return set
  else
    $set = set \cup \{next\}$
    while $N[next] \neq next$ **do** ▷ Iterate till end of list
      $next = N[next]$
      $set = set \cup \{next\}$
    end while
  end if
  return set
end procedure

Algorithm 4. 5 round AES-128 Attack (Partial Key)

Require: $pdiag \in \{0, 1, 2, 3\}$
procedure ATTACKPARTIAL($pdiag$) ▷ $pdiag$ is the index of diagonal of the partial key
  `HashTable` $\leftarrow$ LOADHASHTABLE()
  $IK \leftarrow$ INITIALIZEIK() ▷ IK is the key-array used to keep track of elimination of keys
  for $cdiag = 0$ to 3 **do** ▷ $cdiag$ is the index of the ciphertext diagonal
    `DesiredTable` $\leftarrow$ BUILDDESIREDTABLE($pdiag, cdiag$)
    for $dkey = 0$ to $2^{32} - 1$ **do**
      for all $\{P_1, P_2\} \in$ `DesiredTable`$[dkey]$ **do**
        $hindex \leftarrow P_1 \oplus P_2$
        for all $x \in$ `HashTable`$[hindex]$ **do**
          ELIMINATEKEY($IK, x \oplus P_1$) ▷ The partial key $x \oplus P_1$ is eliminated
          ELIMINATEKEY($IK, x \oplus P_2$)
        end for
      end for
    end for
  end for
  $KeySet \leftarrow$ GETVALIDKEYS(IK) ▷ Get the partial keys which were not eliminated
  return $KeySet$
end procedure

Table 2. Memory usage of implementation

Data structure	Memory used
Hash table	96 GB
Desired table	32 GB
Key array	0.5 GB
Total	128.5 GB

Table 3. Full key recovery attack on 5 round `AES-128`

Key retrieved	Chosen plaintexts	Memory used	Time taken
$K = K_0$	$2^{32} \cdot 4$	128.5 GB	$4 \times 12 = 48$ h

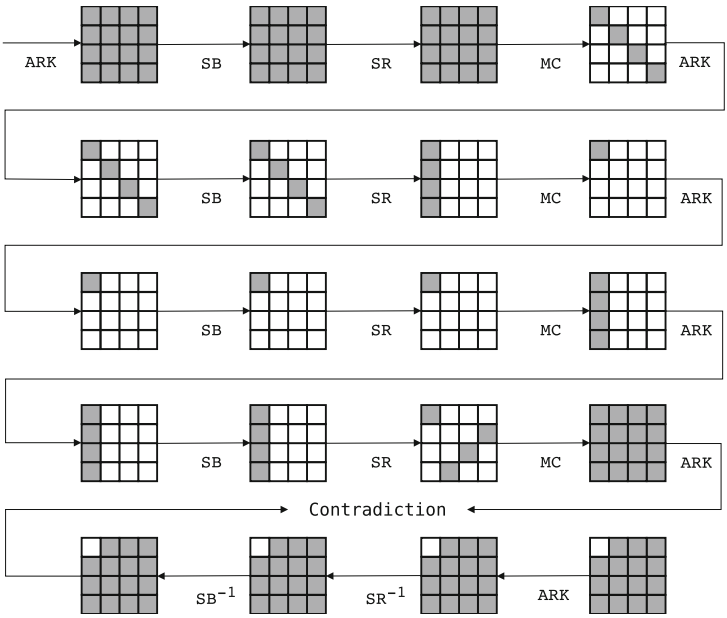


Fig. 6. Path to retrieve K_1 for 5 round `AES-192/256`

need to redefine chosen plaintexts. Chosen plaintexts are plaintexts s_0, whose bytes in the state $s_1^{\mathcal{M}}$ vary along a diagonal from $\mathcal{D}_p$ with other bytes fixed. We choose plaintexts by choosing $s_1^{\mathcal{M}}$ and decrypt back using K_0 to obtain the actual plaintext s_0. The rest of the implementation follows along the same lines of the implementation on `AES-128`. The results are given in Table 4.

Table 4. Full key recovery attack on 5 round `AES-256`

Key retrieved	Chosen plaintexts	Memory used	Time taken
K_0	$2^{32} \cdot 4$	128.5 GB	$4 \times 12 = 48$ h
K_1	$2^{25} \cdot 4$	98 GB	2 h
K	$2^{32} \cdot 4$	128.5 GB	50 h

5 Conclusion

In this work, we have given a data-memory tradeoff for the impossible differential attack on 5 round `AES-128` by Biham and Keller. This data-memory tradeoff enables implementation of the attack with flexible resources to tackle the high memory requirement of 4 TB of the original attack. We have implemented the attack on `AES-128` with a decrease in primary memory from 4 TB to 128.5 GB and increase in data complexity from $2^{29.5}$ to 2^{32} with the time complexity, 2^{31}, not affected. Custom data structures were devised to make the implementation efficient. This attack was extended to `AES-192` and `AES-256` and implemented using the same techniques. This is the first reported implementation of all the attacks mentioned to the best of our knowledge.

This implementation might be useful to future impossible differential attacks on higher number of rounds with practical complexities as the basic primitives of computation in impossible differential attacks are implemented in an efficient way. It might also be useful for gaining insight into the distribution of hashes in the hash table and the key elimination patterns per pair and can be a future extension of this work.

References

1. Bahrak, B., Aref, M.R.: Impossible differential attack on seven-round AES-128. IET Inform. Secur. **2**(2), 28–32 (2008). http://dx.doi.org/10.1049/iet-ifs:20070078
2. Biham, E., Biryukov, A., Shamir, A.: Cryptanalysis of skipjack reduced to 31 rounds using impossible differentials. In: Stern, J. (ed.) EUROCRYPT 1999. LNCS, vol. 1592, pp. 12–23. Springer, Heidelberg (1999). doi:10.1007/3-540-48910-X_2
3. Biham, E., Keller, N.: Cryptanalysis of Reduced Variants of Rijndael. In: 3rd AES Conference, vol. 230 (2002)
4. Biham, E., Shamir, A.: Differential cryptanalysis of DES-like cryptosystems. J. Cryptology **4**(1), 3–72 (1991). http://dx.doi.org/10.1007/BF00630563
5. Mala, H., Dakhilalian, M., Rijmen, V., Modarres-Hashemi, M.: Improved impossible differential cryptanalysis of 7-round AES-128. In: Gong, G., Gupta, K.C. (eds.) INDOCRYPT 2010. LNCS, vol. 6498, pp. 282–291. Springer, Heidelberg (2010). doi:10.1007/978-3-642-17401-8_20
6. Daemen, J., Rijmen, V.: The Design of Rijndael: AES - The Advanced Encryption Standard. Information Security and Cryptography. Springer (2002). http://dx.doi.org/10.1007/978-3-662-04722-4

7. Gueron, S.: Intel® Advanced Encryption Standard (AES) New Instructions Set. Intel Corporation (2010). https://software.intel.com/sites/default/files/article/165683/aes-wp-2012-09-22-v01.pdf
8. Lu, J., Dunkelman, O., Keller, N., Kim, J.: New impossible differential attacks on AES. In: Chowdhury, D.R., Rijmen, V., Das, A. (eds.) INDOCRYPT 2008. LNCS, vol. 5365, pp. 279–293. Springer, Heidelberg (2008). doi:10.1007/978-3-540-89754-5_22
9. Mala, H., Dakhilalian, M., Rijmen, V., Modarres-Hashemi, M.: Improved impossible differential cryptanalysis of 7-round AES-128. In: Gong, G., Gupta, K.C. (eds.) INDOCRYPT 2010. LNCS, vol. 6498, pp. 282–291. Springer, Heidelberg (2010). doi:10.1007/978-3-642-17401-8_20
10. Phan, R.C.: Impossible differential cryptanalysis of 7-round Advanced Encryption Standard (AES). Inf. Process. Lett. **91**(1), 33–38 (2004). http://dx.doi.org/10.1016/j.ipl.2004.02.018