Comprehensive Preimage Security Evaluations on Rijndael-based Hashing

Tianyu Zhang

School of Physical and Mathematical Sciences, Nanyang Technological University, Singapore.
tianyu005@e.ntu.edu.sg

Abstract. The Meet-in-the-Middle (MITM) attack is one of the most powerful cryptanalysis techniques, as seen by its use in preimage attacks on MD4, MD5, Tiger, HAVAL, and Haraka-512 v2 hash functions and key recovery for full-round KTANTAN. An efficient approach to constructing MITM attacks is automation, which refers to modeling MITM characteristics and objectives into constraints and using optimizers to search for the best attack configuration. This work focuses on the simplification and renovation of the most advanced superposition framework based on Mixed-Integer Linear Programming (MILP) proposed at CRYPTO 2022. With the refined automation model, this work provides the first comprehensive analysis of the preimage security of hash functions based on all versions of the Rijndael block cipher, the origin of the Advanced Encryption Standard (AES), and improves the best-known results. Specifically, this work has extended the attack rounds of Rijndael 256-192 and 256-256, reduced the attack complexity of Rijndael 256-128 and 128-192 (AES192), and filled the gap of preimage security evaluation on Rijndael versions with a block size of 192 bits.

Keywords: Rijndael · Preimage · Hashing mode · MITM · MILP

1 Introduction

A hash function constructs a fixed-length message digest for an arbitrary-length plaintext. Hash functions have wide applications in cybersecurity infrastructure, including but not limited to, digital signatures, fingerprinting, authentication schemes, commitment schemes, and error correction codes. To be cryptographically secure, a hash function must satisfy the three fundamental security requirements: preimage resistance, second preimage resistance, and collision resistance.

A common strategy for building hash functions follows a two-step approach: first, a compression function is formed by inserting a block cipher into a PGV mode [2]; then, a hash function is constructed by iterating the compression function following the Merkle-Damgård paradigm (Fig. 1). It is proven that the resulting hash function enjoys security reduction to the underlying encryption. Typical choices for the PGV mode include Davies-Meyer (DM), Matyas-Meyer-Oseas (MMO), and Miyaguchi-Preneel (MP) (Fig. 2). The strategy is highly practical since block ciphers and hash functions often coexist in an information system. The cost of implementing an additional hash function is minimized by applying only an extra mode to the already built-in block cipher.

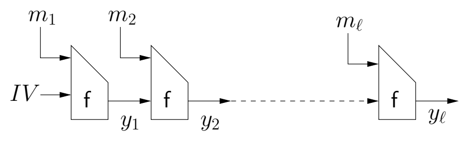
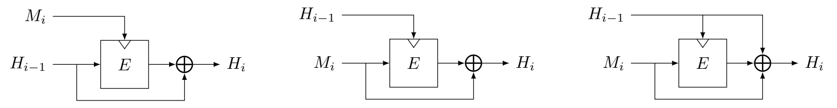


Fig. 1: Merkle-Damgård construction [45].



(a) DM-mode (PGV No.5) $H_i = E_{M_i}(H_{i-1}) \oplus H_{i-1}$

(b) MMO-mode (PGV No.1) $H_i = E_{H_{i-1}}(M_i) \oplus M_i$

(c) MP-mode (PGV No.3) $H_i = E_{H_{i-1}}(M_i) \oplus M_i \oplus H_{i-1}$

Fig. 2: Diagrams for DM, MMO, and MP modes [30].

This work focuses on the preimage security of hash functions built on the Rijndael block cipher. In 2001, NIST selected versions of the Rijndael block cipher with a block size of 128 bits for the Advanced Encryption Standard (AES) [1]. The MMO mode instantiated with AES has been standardized by Zigbee [3] and ISO [31] for building hash functions on block ciphers. The excellent performance and high security of Rijndael motivate dedicated designs built with a similar structure, for instance, Whirlpool [9], Grøstl [32], PHOTON [33] and LED [34].

1.1 The Meet-In-The-Middle (MITM) Technique

MITM is a well-referenced and well-developed cryptanalysis technique for preimage security analysis. The concept of MITM attacks originated from Diffie and Hellman's time-memory trade-off on double encryption [35]. Since its introduction, advanced techniques have been added to the MITM framework to exploit more freedom and structures, for instance, internal state guesses [36], splice-and-cut [16, 21], initial structures [19] and indirect partial-matching [18, 20].

Pioneered by Aumasson et al. [5] and Sasaki et al. [18, 20], MITM has demonstrated its power in the preimage security analysis of hash functions, including but not limited to MD4 [16], MD5 [19], Tiger [16, 24], HAVAL [27] and KANTAN [10, 25]. The intuition of MITM is to divide the compression function into two independent chunks that "meet in the middle" at some matching point. A MITM attack feeds candidate texts into the two chunks independently and filters the preimage space according to the matching conditions.

In 2011, Sasaki et al. [17] mounted the first MITM preimage attack on AES-hashing. The attack circumvented the AES key schedule by fixing round keys to constants and reached 7 rounds for all AES versions. In [6], Bao et al. revisited the attack and retrieved degrees of freedom from the key schedule. The observation led to improvements in the attack rounds of AES-192 and AES-256, from 7 rounds to 8 rounds. In 2021, [7] drove MITM into automation with Mixed-Integer Linear Programming (MILP). The automatic model generalized and incorporated all enhancing tools of MITM attack, and improved 1 attack round for all best-known attacks on AES (8 rounds AES-128, 9 rounds AES-192, and 9 rounds AES-256). The automation model was then renovated in 2022 with a novel superposition framework that parallels linear operations [8]. The paper also presented new technologies to further empower the model and enlarge the search space, including Guess-and-Determine (GnD), Multiple Ways of AddRoundKey (MulAK), and Bi-Directional Attribute-Propagation and Cancellation (BiDir).

1.2 Contributions

While the preimage security of AES-hashing is a widely concerned and repeatedly investigated proposition in symmetric-key cryptanalysis, the resistance of hashing modes built on other versions of Rijndael against preimage attacks is rarely explored. This paper is thus dedicated to initiating the first comprehensive and comparative study of the preimage security of Rijndael-based hashing using MITM attacks, with a refined and enhanced automation model. Contributions include:

Lightweight Model. The automatic search of MITM attacks is extremely time-demanding and can only be partially optimized. Hence, model simplification has been a persistent endeavor for the community to unleash the full potential of automation. In 2022, the superposition structure [8] introduced two virtual states to separate forward and backward propagation for each intermediate state that undergoes linear operators. However, redundancies persist, since the encoding scheme used was originally proposed to address both propagations in one intermediate state. This work provides a dedicated lightweight encoding scheme for the superposition structure that enables simpler modeling of propagation rules, thus laying a solid foundation for the automation model to incorporate more techniques.

Mega-MC Match. In this paper, a new match rule, namely MEGA-MC-MATCH, is proposed to extend the traditional matching through the `MixColumns` operator at intermediate rounds. The MEGA-MC-MATCH further exploits the properties of the diffusion matrix and utilizes information that was previously considered ineligible for matching. The naming of the new match originates from its ability to extend the matching point from a single `MixColumns` operator to a 'mega'-variant that involves 3 rounds.

Accurate Key Schedule. There have been unaddressed dependencies in Rijndael's key schedule that might lead to inaccurate propagation patterns. In this work, such dependencies are identified, and a more accurate model of the Rijndael key schedule is provided. The new model is able to provide adequate treatment to equivalencies and dependencies in the key schedule and prevent repeated consumption of degrees of freedom.

Summary of application results. This work has achieved the following accomplishments: extended the maximum attack rounds of Rijndael 256-192 and 256-256 by 1 round, reduced the time complexity of 8-round Rijndael 128-192 (AES192) and 9-round Rijndael 256-128, and filled the gap of preimage security evaluation on Rijndael versions with 192-bit block size. Detailed results are shown in Table 1.

Table 1: Updated results on pseudo-preimage and preimage attacks.

Block length	Key length	Rounds	Pseudo-preimage	Preimage	Ref.
128	128	8/10	2^{120}	2^{125}	[7]
	192	8/12	2^{112}	2^{116}	[6]
		8/12	2^{104}	2^{117}	Fig. 7
		9/12	2^{112}	2^{121}	[8]
	256	10/14	2^{120}	2^{125}	[7]
192	128	9/12	2^{184}	2^{189}	Fig. 8
	192	9/14	2^{184}	2^{189}	Fig. 9
	256	9/14	2^{176}	2^{185}	Fig. 10
256	128	9/14	2^{248}	2^{253}	[7]
		9/14	2^{240}	2^{249}	Fig. 11
	192	9/14	2^{248}	2^{253}	[7]
		10/14	2^{248}	2^{253}	Fig. 12
	256	9/14	2^{248}	2^{253}	[7]
		10/14	2^{248}	2^{253}	Fig. 6

2 Preliminaries

2.1 The Rijndael Block Cipher

Versions. Rijndael is a family of iterated block ciphers developed by Belgian cryptographers Joan Daemen and Vincent Rijmen with different key sizes and block sizes [1]. Rijndael supports a variety of combinations of block sizes and key sizes, which can be specified independently as 128, 192, or 256 bits. the intermediate result in the encryption is denoted as a *state* and the cipher key as the *key*. Intuitively, a *state* and the *key* can each be pictured as a 4-rowed rectangular array with 1 byte per entry. Conventionally, a column is referred to as a word, and the word size is thus fixed to 32 bits. *Nb* denotes the number of words in a *state* and *Nk* denotes that of the *key*. *Nr* denotes the iterated rounds, which is dependent on *Nb* and *Nk* (Table 2). In 2001, NIST selected the versions of Rijndael with a 128-bit block length as the new symmetric key encryption standard (AES).

Round Operators. The Rijndael round function consists of 4 different operators:

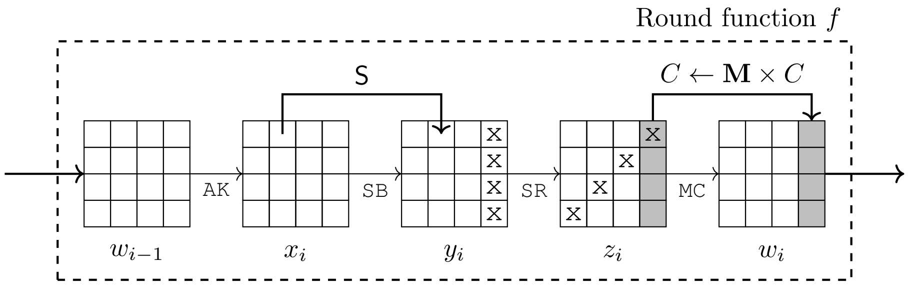


Fig. 3: Rijndael round function.

`SubBytes`: A non-linear byte-wise substitution taking 1 byte as input and producing 1 byte as output. The input and the output byte are interpreted as polynomials on $GF(2^8)$ in their vector form. The S-Box contains two steps: the input a is first mapped to its multiplicative inverse a^{-1} in $GF(2^8)$. Then, a^{-1} is mapped to the output s by an affine transformation:

$$b = a^{-1} \oplus (a^{-1} \lll 1) \oplus (a^{-1} \lll 2) \oplus (a^{-1} \lll 3) \oplus (a^{-1} \lll 4) \oplus 63_{16}$$

`ShiftRows`: A linear transformation, visualized as a circular left shift on the rectangular array. The shift offsets for each row are dependent on *Nb*, as shown in Table 3.

`MixColumns`: A column-wise linear transformation, described as a left multiplication in $GF(2^8)$ with a constant 4-by-4 maximum distance separable (MDS) matrix.

`AddRoundKey`: The bitwise XOR of the round key and the current state.

Key Schedule. The round keys are derived from the *key* by `KeySchedule` with two steps:

`KeyExpansion` The *key* is first expanded to an array of bytes w with $|w| = 4 \cdot Nb \cdot (Nr + 1)$. When $i < Nk$, $w[i] = key[i]$. Otherwise, the expansion follows the following equation, where *Rcon* is a constant array, and `Rot` is a permutation on the bytes:

$$w[i] = \begin{cases} w[i - Nk] \oplus \texttt{SubBytes}(\texttt{Rot}(w[i-1])) \oplus Rcon[i/Nk] & i \bmod Nk \equiv 0 \\ w[i - Nk] \oplus \texttt{SubBytes}(w[i-1]) & i \bmod Nk \equiv 4 \text{ and } Nk = 8 \\ w[i - Nk] \oplus w[i-1] & \text{otherwise} \end{cases} \tag{1}$$

`RoundKeySelection` The round keys, each consisting of *Nb* words, are taken sequentially from *w*.

Although the round keys are generated one byte at a time, for easier illustration, a `KeySchedule` round is defined as the period when `KeyExpansion` produces an additional *Nk* words. The key schedule will iterate a total of $Nb * (Nr + 1)/Nk$ rounds to generate all round keys. Each byte of a round key is uniquely indexed by (r, i, j) similar to encryption states, where r denotes the key schedule round, and (i, j) denotes the position of the cell on the key grid.

Rijndael Encryption. Algorithm 1 shows the pseudocode for the encryption process with Rijndael. The `MixColumns` operator in the last round is omitted, resulting in a more symmetric structure and allowing efficient conversion from encryption to decryption. The cipher starts and ends with an `AddRoundKey` operation. This particular design is called "key whitening", which conceals the information immediately after the iteration starts and before the iteration ends.

2.2 Preimage and pseudo-preimage attacks

A preimage attack finds a preimage of a given digest y for a hash function $\mathcal{H}$:

$$\text{Given } \mathcal{H} : X \to Y, \text{ for } y \in Y, \text{ find } x \in X, \text{ s.t. } \mathcal{H}(x) = y$$

For hash functions built with the Merkel-Damgård construction, a pseudo-preimage attack focuses on the underlying compression function. It finds a pair of messages x and a chaining value H that leads to the target digest y by the compression function $\mathcal{CF}$:

$$\text{Given } \mathcal{CF} : X \to Y, \text{ for } y \in Y, \text{ find } x \in X \text{ and } H \in Y, \text{ s.t. } \mathcal{CF}(H, x) = y$$

2.3 The MITM pseudo-preimage attack

The MITM technique is, in essence, used for finding pseudo-preimages. It makes use of the loop structure determined by PGV modes and divides a hash function into two independent functions, forward and backward. In each function, the involved bytes are categorized as:

Algorithm 1: Rijndael Encryption

```
Input: $m$, $key$
Output: $c$
/* generation of round keys */
1  $roundkeys \leftarrow$ KeySchedule($key$)
/* initialization */
2  $state \leftarrow m$
3  $state \leftarrow$ AddRoundKey($state, roundkeys[0]$)
4  for $r \leftarrow 0$ to $Nr - 2$ do
5      $state \leftarrow$ SubBytes($state$)
6      $state \leftarrow$ ShiftRow($state, Nb$)
7      $state \leftarrow$ MixColumns($state$)
8      $state \leftarrow$ AddRoundKey($state, roundkeys[r]$)
9  $state \leftarrow$ SubBytes($state$)
10 $state \leftarrow$ ShiftRow($state, Nb$)
11 $c \leftarrow$ AddRoundKey($state, roundkeys[Nr]$)
```

– neutral bytes, whose values are only known in the current function and have no influence on the other function.

– constant bytes, whose values are predefined and globally known in both functions.

The two functions meet structurally in a shared intermediate state called the matching point. Constraints invoked at the matching point for the integrity of the closed loop are called partial-match constraints, which will be exploited as a filter to eliminate ineligible candidates. A full match check is performed only when partial-match constraints are satisfied. The generic MITM attack framework used in multiple references is reiterated as follows [6–8, 37]:

1. Assign random values to constant bytes.
2. Determine the candidate values for the neutral bytes N_+ and N_-. Assume that there are 2^{d_1} candidates for N_+ and 2^{d_2} candidates for N_-.
3. For each of the 2^{d_1} candidates of N_+, compute the forward function and store the output at the matching point in a table T_+.
4. For each of the 2^{d_2} candidates of N_-, compute the backward function and store the output at the matching point in a table T_-.
5. Assume that there are 2^m constraints at the matching point. For the indices of T_+ and T_-, select pairs satisfying the 2^m partial-match constraints.
6. For the survived pairs, check for a full match.

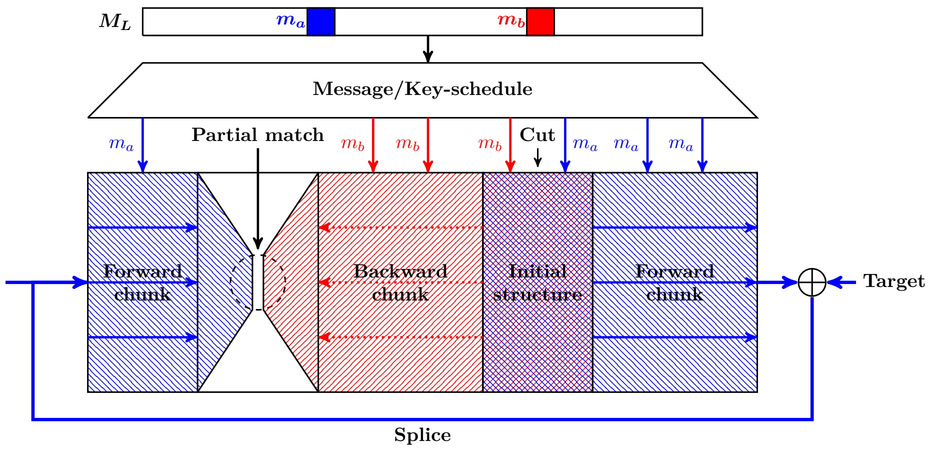


Fig. 4: The latest MITM pseudo preimage attack framework [17].

7. If there exists a full match, a pseudo-preimage is found. Otherwise, revert to Step 1, change the arbitrary values, and repeat procedures 2 to 6.

The computational complexity of the attack is calculated as follows [6]:

$$2^{n-(d_1+d_2)} \cdot (2^{max(d_1,d_2)} + 2^{d_1+d_2-m}) \simeq 2^{n-min(d_1,d_2,m)} \tag{2}$$

2.4 Pseudo-preimage to preimage conversion

A pseudo-preimage attack with a computational complexity of 2^l ($l < n - 2$) can be converted to a preimage attack with a computation complexity of $2^{(n+l)/2+1}$ [43]. To achieve this, $2^{(n-l)/2}$ pseudo-preimages are found. Next, starting from the initialization vector IV, $2^{(n+l)/2+1}$ random values are inserted into the hash function to generate $2^{(n+l)/2+1}$ chaining values. With $2^{(n+l)/2+1}$ chaining values mapped from the real IV and $2^{(n-l)/2}$ pseudo-preimages mapped to the target hash value, a match can be expected thanks to the birthday bound.

In 2008, Leurent improved the general unbalanced meet-in-the-middle method by constructing an unbalanced tree using multi-target pseudo-preimage and using the expandable message technique to overcome the length padding [44]. The overall time complexity is improved to $(n \ln 2 - l \ln 2 + 1) \cdot 2^l$. However, the method assumes a special unbalanced condition where 2^{l-t} computations yield 2^t pseudo-preimages.

3 MILP Modeling for Automated Search

This section describes how the search for MITM preimage attacks on Rijndael is automated with Mixed-Integer Linear Programming (MILP). This paper follows the conventional MITM coloring schemes for visualizing and demonstrating results [6–8, 17, 37]. A byte or a cell in an intermediate state or a round key is colored as follows:

– A blue cell (🟦) denotes a neutral byte for the forward function.

– A red cell (🟥) denotes a neutral byte for the backward function.

– A gray cell (⬛) denotes a constant byte.

– A white cell (⬜) denotes an arbitrary byte, incomputable in both functions.

3.1 Automated Search Framework

An overview of the automatic search framework of MITM attacks is provided before digging into the details. The essence of a MITM attack lies in the careful segmentation of the closed computation path. The special states in a MITM attack are identified using the following notations:

– $\overleftrightarrow{S}^{\texttt{ENC}}$: the starting encryption state for forward and backward functions.

– $\overleftrightarrow{S}^{\texttt{KSA}}$: the starting key schedule state for forward and backward functions.

– $\overrightarrow{End}$: the terminating state in encryption for the forward function.

– $\overleftarrow{End}$: the terminating state in encryption for the backward function.

– $\overleftrightarrow{M}^{\texttt{Match}}$: the matching round operator between the two terminating states.

The locations of the special states mentioned above represent different ways of segmenting the closed loop, hence uniquely determining the structure of a MITM attack. The locations are modeled with round-level precision. The attack configuration parameter *config* is defined as the ordered tuple with the following attributes:

– *Total*: the total attacked rounds.

– *EncSt*: the round index of $\overleftrightarrow{S}^{\texttt{ENC}}$.

– *KeySt*: the round index of $\overleftrightarrow{S}^{\texttt{KSA}}$.

– *Match*: the round index of $\overleftrightarrow{M}^{\texttt{Match}}$, $\overrightarrow{End}$ and $\overleftarrow{End}$.

To enable independent computations, the bytes (cells) in $\overleftrightarrow{S}^{\texttt{ENC}}$ and $\overleftrightarrow{S}^{\texttt{KSA}}$ are partitioned into subsets with different coloring $\mathcal{B}^{\texttt{ENC}}$, $\mathcal{R}^{\texttt{ENC}}$, $\mathcal{G}^{\texttt{ENC}}$ and $\mathcal{B}^{\texttt{KSA}}$, $\mathcal{R}^{\texttt{KSA}}$, $\mathcal{G}^{\texttt{KSA}}$ satisfying the following relations:

$$\begin{aligned}\mathcal{B}^{\texttt{ENC}} \cup \mathcal{R}^{\texttt{ENC}} \cup \mathcal{G}^{\texttt{ENC}} &= \{0, 1, \ldots, Nb\}\\ \mathcal{B}^{\texttt{ENC}} \cap \mathcal{R}^{\texttt{ENC}} &= \varnothing\\ \mathcal{R}^{\texttt{ENC}} \cap \mathcal{G}^{\texttt{ENC}} &= \varnothing\\ \mathcal{G}^{\texttt{ENC}} \cap \mathcal{B}^{\texttt{ENC}} &= \varnothing\end{aligned} \tag{3}$$

$$\begin{aligned} \mathcal{B}^{\mathtt{KSA}} \cup \mathcal{R}^{\mathtt{KSA}} \cup \mathcal{G}^{\mathtt{KSA}} &= \{0, 1, \dots, Nk\} \\ \mathcal{B}^{\mathtt{KSA}} \cap \mathcal{R}^{\mathtt{KSA}} &= \varnothing \\ \mathcal{R}^{\mathtt{KSA}} \cap \mathcal{G}^{\mathtt{KSA}} &= \varnothing \\ \mathcal{G}^{\mathtt{KSA}} \cap \mathcal{B}^{\mathtt{KSA}} &= \varnothing \end{aligned} \tag{4}$$

The initial degrees of freedom of forward and backward computations are denoted by $\overleftarrow{\iota}$ and $\overrightarrow{\iota}$. In the attribute propagation of each function, additional constraints may be imposed to cancel mutual impact and preserve functional independence. The consumed degrees of freedom (DOFs) are denoted as $\overrightarrow{\sigma}$ and $\overleftarrow{\sigma}$. The remaining DOFs at the end of each computation are denoted as $\overrightarrow{d_b}$ and $\overleftarrow{d_r}$. Relations can be formulated intuitively as follows:

$$\begin{aligned} \overrightarrow{\iota} &= |\mathcal{B}^{\mathtt{ENC}}| + |\mathcal{B}^{\mathtt{KSA}}| \\ \overleftarrow{\iota} &= |\mathcal{R}^{\mathtt{ENC}}| + |\mathcal{R}^{\mathtt{KSA}}| \end{aligned} \tag{5}$$

$$\begin{aligned} \overrightarrow{d_b} &= \overrightarrow{\iota} - \overrightarrow{\sigma} \\ \overleftarrow{d_r} &= \overleftarrow{\iota} - \overleftarrow{\sigma} \end{aligned} \tag{6}$$

The distribution of $\overleftrightarrow{M}^{\mathtt{Match}}$, $\overrightarrow{End}$, and $\overleftarrow{End}$ decides the degree of matching $\overset{\rightarrow\leftarrow}{m}$. According to Equation 2, $\min\{\overrightarrow{d_b}, \overleftarrow{d_r}, \overset{\rightarrow\leftarrow}{m}\}$ determines the complexity of a MITM attack. Thus, the search for the optimal MITM attack pattern of given *config* is converted to a maximization problem on objective $\tau_{\mathtt{Obj}}$:

$$\begin{aligned} \max_{config} \quad & \tau_{\mathtt{Obj}} \\ \text{s.t.} \quad & \tau_{\mathtt{Obj}} \leq \overrightarrow{d_b} \\ & \tau_{\mathtt{Obj}} \leq \overleftarrow{d_r} \\ & \tau_{\mathtt{Obj}} \leq \overset{\rightarrow\leftarrow}{m} \\ & \tau_{\mathtt{Obj}} > 0 \end{aligned} \tag{7}$$

Given an n-bit target, the pseudo-preimage attack complexity of an attack configuration $(\overrightarrow{d_b}, \overleftarrow{d_r}, \overset{\rightarrow\leftarrow}{m})$ will be (in the exponent of 2):

$$n - \min\{\overrightarrow{d_b}, \overleftarrow{d_r}, \overset{\rightarrow\leftarrow}{m}\}$$

A preimage attack can be constructed based on a pseudo-preimage attack with time complexity (in the exponent of 2) as follows:

$$\begin{cases} n - \min(\overrightarrow{d_b}, \overleftarrow{d_r}, \overset{\rightarrow\leftarrow}{m}) + \log_2(\min(\overrightarrow{d_b}, \overleftarrow{d_r}) \ln 2 + 1) & \text{if } \min(\overrightarrow{d_b}, \overleftarrow{d_r}) < \overset{\rightarrow\leftarrow}{m} \\ n - \min(\overrightarrow{d_b}, \overleftarrow{d_r}, \overset{\rightarrow\leftarrow}{m})/2 + 1 & \text{otherwise} \end{cases} \tag{8}$$

3.2 The Superposition State Structure

For an intermediate state around linear operators, the superposition structure [8] introduces two superposition states, each carrying the propagation of only one function. The rationale behind the separation is due to the linearity of the operators. An intermediate state and its superposition states are denoted as s, s_F, and s_B. If there is a bilinear function λ such that $s = \lambda(s_F, s_B)$, then s propagating through a Rijndael linear operator χ is expressed as $\chi(s) = \chi(\lambda(s_F, s_B))$. For deterministic bilinear function $\lambda' = \chi\lambda(\chi^{-1} \times \chi^{-1})$, $\chi(s)$ can be expressed as $\chi(s) = \lambda'(\chi(s_F), \chi(s_B))$. Hence, the propagation of the intermediate state is equivalent to propagating two superposition states independently if the superposition states are bilinearly associated before. However, such parallel propagation must end when undergoing non-linear operators where the linear relation between two propagation trails will be destroyed.

Specifically, the superposition structure is deployed in Rijndael with the following heuristic:

1. The single state separates into two superposition states after `SubBytes`.
2. The superposition states propagate independently through other operators.
3. The superposition states collapse to a single state.
4. The single state propagates through `SubBytes`.

The superposition technique automatically supports the BiDir technique [8], which generalizes DOF consumptions in forward and backward computations.

3.3 Simplified Encoding Scheme

The conventional encoding scheme [7, 8] used two encoders x and y to encode different coloring: ⬜ $= (0, 0)$, 🟦 $= (1, 0)$, 🟥 $= (0, 1)$, ⬛ $= (1, 1)$. To uniquely identify ⬜ and ⬛, two additional encoders are used:

$$\begin{aligned} g &= x \vee y \\ w &= 1 + g - x - y \end{aligned} \tag{9}$$

The $x - y - g - w$ encoding allows redundancies in the superposition structure. First, the scheme was originally introduced to cover two propagations simultaneously at a single state. However, in superposition states, attribute propagation no longer requires information from both functions. In other words, 🟦 will never appear in the backward computation and 🟥 in the forward computation. Second, the identification of ⬜ and ⬛ (g and w encoders) is complex in implementation and difficult for preprocessing.

In this work, a symmetric and efficient encoding scheme is proposed dedicated to the superposition structure. The scheme involves two variables α and β:

– α: equals 1 if and only if the byte could be calculated in the current function.

– β: equals 1 if and only if the exact value of the byte is known in the current function.

It is easy to observe that $\beta = 1$ is a stronger condition than $\alpha = 1$. Therefore, $\beta \leq \alpha$ is enforced. The propagation is symmetrically encoded as: 🟦 or 🟥 $(\alpha, \beta) = (1, 0)$, ▨ $(\alpha, \beta) = (1, 1)$ and □ $(\alpha, \beta) = (0, 0)$. In this encoding scheme, $\beta = 1$ uniquely identifies ▨, and $\alpha = 0$ uniquely identifies □.

Due to the symmetric nature of the encoding, in subsequent sections, the MILP engraving of propagation rules will only be detailed for the forward computation. The rules for backward computation are autonomous.

Modelling the start and end of superposition states. EnterSUP-Rule describes the separation of a single state $s = (x, y)$ into two superposition states $s_F = (\alpha_F, \beta_F)$ and $s_B = (\alpha_B, \beta_B)$ with the $\alpha - \beta$ encoding. The separation is performed byte-wise: If a byte s is □ or ▨, then s_F and s_B are both arbitrary or constant, either way, s_F and s_B share the same coloring as s. Suppose s is 🟦 or 🟥, which means that the color of s is preserved in the corresponding computation and the other direction will be compensated with ▨ symbolically, indicating a constant influence:

$$\begin{aligned}(\alpha_F, \beta_F) &= (x \vee y, y)\\ (\alpha_B, \beta_B) &= (x \vee y, x)\end{aligned} \tag{10}$$

Before `SubBytes`, the ExitSUP-Rule collapses the two virtual states $s_F = (\alpha_F, \beta_F)$ and $s_B = (\alpha_B, \beta_B)$ into a single state $s = (x, y)$ before `SubBytes`. If the collapsed state is influenced by an arbitrary byte or by both the forward and the backward computations, the state is arbitrary. The rule is formulated as follows:

$$(x, y) = (\alpha_F \wedge \beta_B, \alpha_B \wedge \beta_F) \tag{11}$$

Modelling `SubBytes` and `ShiftRows`. The `SubBytes` operator itself does not change the attribute of the cells. As long as the superposition states are collapsed properly before `SubBytes`, the operator does an identity transformation on the coloring. The `ShiftRows` operator permutes the state cells according to some predefined constants and thus can be modeled by a set of equalities between dedicated variables or some hardcoded variable substitutions, both of which are intuitive and autonomous.

Modelling `AddRoundKey`. XOR is the basic operator in `AddRoundKey`, which takes two cells as input and outputs one cell. Under the $\alpha - \beta$ encoding, the XOR-Rule could be simplified compared to previous works [7, 8]. The rule is described as follows:

– When the inputs contain □, the output is □.

– When the inputs are both ▨, the output is ▨.

– Otherwise, the output is:

  - 🟦, with no consumption of DOF.
  - ▨, consuming 1 DOF of the forward computation.($\overrightarrow{\sigma} = \overrightarrow{\sigma} + 1$)

The above description is converted to constraints by the convex-hull method [23]. The resulting inequalities should involve the following variables: the (α, β)-encoders of both inputs and the output together with an indicator variable to track DOF consumptions.

Modelling the `KeyExpansion`. KeyXOR-Rule is introduced to identify and address dependencies in the `KeySchedule`. Recall Equation 1, a node $w[i]$ in `KeyExpansion` has two parents: $w[i-1]$ and $w[i-Nk]$. If the index i satisfies the condition:

$$i \not\equiv \begin{cases} 0 & Nk \leq 6 \\ 0,4 & Nk > 6 \end{cases}, \quad \pmod{Nk} \tag{12}$$

$w[i] = w[i-Nk] \oplus w[i-1]$. If the same condition holds for $i-1$, $w[i]$ can be expressed using only $w[i-2Nk]$ and $w[i-2]$:

$$\begin{aligned} w[i] &= w[i-Nk] \oplus w[i-1] \\ &= w[i-2Nk] \oplus w[i-Nk-1] \oplus w[i-Nk-1] \oplus w[i-2] \\ &= w[i-2Nk] \oplus w[i-2] \end{aligned} \tag{13}$$

Note that the middle term $w[i-Nk-1]$ cancels due to the consecutive XORs without confusion. However, since the basic XOR-Rule sequentially obtains the parents of $w[i]$ and then $w[i]$ itself, the coloring of $w[i-Nk-1]$ will still affect $w[i]$. For instance, if $w[i-Nk-1]$ is □ while $w[i-2Nk]$ and $w[i-2]$ are 🟦 or ▨, then $w[i]$ will be miscolored to □. Moreover, if $w[i-2Nk]$ and $w[i-2]$ are both ▨ and $w[i-Nk-1]$ is 🟦, $w[i]$ will be miscolored as 🟦 or waste an unnecessary DOF to cancel impact by turning ▨. To address such dependencies, two additional encoders α_{eq} and β_{eq} are introduced for KeyXOR-Rule:

$$\begin{aligned} \alpha_{eq} &= \begin{cases} \min(\alpha^{w[i-2Nk]}, \alpha^{w[i-2]}) & i, i-1 \text{ s.t. condition 12} \\ 0 & \text{otherwise} \end{cases} \\ \beta_{eq} &= \begin{cases} \min(\beta^{w[i-2Nk]}, \beta^{w[i-2]}) & i, i-1 \text{ s.t. condition 12} \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{14}$$

Due to the enforced constraint $\beta \leq \alpha$, it follows that $\beta_{eq} \leq \alpha_{eq}$. The KeyXOR-Rule is defined as follows:

- When $\beta_{eq} = 0$, and $\alpha_{eq} = 0$, apply the XOR-Rule with respect to $w[i-1]$ and $w[i-Nk]$
- When $\beta_{eq} = 0$, and $\alpha_{eq} = 1$, apply the XOR-Rule with respect to $w[i-1]$ and $w[i-Nk]$, but override the coloring of the output cell whenever the inputs contain ☐ as follows:
  - 🟦, without consumption of DOF.
  - ■, consuming 1 DOF from the forward computations. ($\overrightarrow{\sigma} = \overrightarrow{\sigma} + 1$)
- When $\beta_{eq} = 1$, and $\alpha_{eq} = 1$, the output is ■ without DOF consumption.

The above description is also translated into constraints by the convex-hull method. The inequalities involve the (α, β) encoders of the two inputs and one output, the equivalence encoders α_{eq}, β_{eq}, as well as an indicator variable to track DOF consumption.

Modeling `MixColumns`. A `MixColumns` operator takes a column as input and outputs a column. The propagation rule is described as follows:

- When the inputs contain ☐, the outputs are all ☐.
- When the inputs are all ■, the outputs are all ■.
- Otherwise, the output will be (WLOG, in forward computation):
  - 4 blue cells (🟦), without consumption of DOF.
  - b blue (🟦) cells and g gray (■) cells, with $b+g = 4$ and $g > 1$, consuming forward DOF(s) [7].

In [7, 8], realizing above rules with $x-y-g-w$ encoding requires three additional columnwise encoders μ,ν,ω. The input column is superscripted I and the output column O. The exact implementation is shown as follows:

$$\begin{cases} \sum_{i=0}^{3} x_i^O + 4 \cdot \omega \leq 4 \\ \sum_{i=0}^{3} (x_i^I + x_i^O) - 8 \cdot \mu \geq 0 \\ \sum_{i=0}^{3} (x_i^I + x_i^O) - 5 \cdot \mu \leq NIO - Br \\ \sum_{i=0}^{3} y_i^O + 4 \cdot \omega \leq 4 \\ \sum_{i=0}^{3} (y_i^I + y_i^O) - 8 \cdot \nu \geq 0 \\ \sum_{i=0}^{3} (y_i^I + y_i^O) - 5 \cdot \nu \leq 3 \\ \sum_{i=0}^{3} x_i^O - 4 \cdot \mu = cost_F \\ \sum_{i=0}^{3} y_i^O - 4 \cdot \nu = cost_B \end{cases} \tag{15}$$

The modeling of the `MixColumns` operator can be simplified using the new encoding scheme. Only two additional encoders κ and ψ are introduced to provide quick identification of a column being all ⬛ or existing ⬜:

$$\begin{aligned} \kappa &= \min_i(\alpha_i^I) \\ \psi &= \min_i(\beta_i^I) \end{aligned} \tag{16}$$

By definition, $\kappa = 0$ if and only if there is ⬜ among the input, and $\psi = 1$ if and only if the inputs are all ⬛. The MC-Rule is thus defined minimally:

$$\begin{aligned} \alpha_i^O &= \kappa \\ \beta_i^O &= \psi + \varsigma_i \end{aligned} \tag{17}$$

The binary cost variables ς_i are byte-wise, as an indication of whether a cost of DOF occurs locally for a single cell, instead of integer-valued $cost_F$ and $cost_B$ tracking the total cost for the whole column in previous models [7].

To achieve maximum speedup, the modeling does not provide extra treatment in special scenarios where the actual cost of DOF is less than $\sum_i \varsigma_i$. For example, if the inputs consist of 1 🟦 and 3 ⬛, then the outputs will be all ⬛ with 1 DOF cost, as the inputs only have 1 DOF. In the above modeling, the total cost will still be 4 since the cost of DOF is counted byte-wise. However, since the MILP model is globally optimized, such a propagation pattern in this model will be equivalent to turning the only 🟦 in the `MixColumns` inputs to ⬛ during the last `AddRoundKey` operator, consuming 1 DOF. By doing so, the input column of `MixColumns` will be all ⬛ and the global DOF cost remains the same. Hence, the special scenario is neglected to maintain minimal construction.

Modeling the matching. There are three types of matching rules used in this work:

– ID-Match: identity match in the last round.

– MC-Match: match through a single `MixColumns` operator.

– MegaMC-Match: match through a mega-`MixColumns` operator.

Following traditional notations, the cell locates at the i-th row and the j-th column is indexed by $n = 4 \cdot j + i$.

The ID-Match happens checks $\overrightarrow{End}$ and $\overleftarrow{End}$ byte by byte in single states, a matching happens when at index n $\overrightarrow{End}[n]$ and $\overleftarrow{End}[n]$ are not ⬜.

The MC-Match checks $\overrightarrow{End}$ and $\overleftarrow{End}$ column by column in superposition states: $\overrightarrow{End}_F$, $\overrightarrow{End}_B$, $\overleftarrow{End}_F$, $\overleftarrow{End}_B$. $\zeta_{i,j} = 1$ if and only if at index $n = 4 \cdot j + i$ the forward branch and backward branch are both non-arbitrary ($\overrightarrow{End}_F[n], \overrightarrow{End}_B[n] \in$ {🟦, 🟥, ⬛}). Due to the fact that the `MixColumns` operator has branch number 5, a linear constraint can be constructed when the input and output columns contain 5 eligible bytes,[7]. And one more linear constraint can always be constructed with one more eligible byte than 5. Hence, the rules for MC-Match are formulated as follows: a match occurs in column k if: $\sum_{i=1}^{4} \zeta_{i,k} > 4$ with matching degree $m_k = \sum_{i=1}^{4} \zeta_{i,k} - 4$.

The MegaMC-Match is proposed as an extension of MC-Match. In the automatic search for long-round MITM attacks, the bottleneck often lies in $\overset{\rightarrow\leftarrow}{m}$. The intention of the new match is to exploit ☐ for matching, which was deemed impossible in previous works, and to increase $\overset{\rightarrow\leftarrow}{m}$. $\overrightarrow{End}$ and $\overleftarrow{End}$ are investigated column by column in single states. The input and output columns are denoted X and Y. Then the `MixColumns` operator could be expressed as follows:

$$\mathcal{M}X = Y \tag{18}$$

Equation 18 is equivalent to 19 if X and Y are viewed as inputs:

$$[\mathcal{M}| - I_4][X|Y] = \mathbf{0} \tag{19}$$

Clearly, the $4 * 8$ matrix $[\mathcal{M}| - I_4$ has a rank 4. Thus, if there exists 4 🟦, the exact values of X and Y are known in the forward computation regardless of the coloring. The forward computation can be reverted back from $\overleftrightarrow{M}^{\texttt{Match}}$ and the matching can be extended to the last superposition states on both sides of $\overleftrightarrow{M}^{\texttt{Match}}$, as illustrated in the diagram below:

$$\begin{tikzcd} X'_F, X'_B \arrow[r, "SupP"] & X' \arrow[r, "\texttt{SubBytes}_1", "\texttt{ShiftRows}"'] & X \arrow[r, leftrightarrow, "match"] & Y & Y' \arrow[l, "\texttt{SubBytes}_2"', "\texttt{AddRoundKey}"] & Y'_F, Y'_B \arrow[l, "SupP"'] \end{tikzcd}$$

The matching is equivalently considered as the match between X' and X'_F, X'_B through $\texttt{SubBytes}_1$, and between Y' and $Y'F, Y'B$ through $\texttt{SubBytes}_2$. For instance, in Fig. 5, due to the 4 🟦 in $\overrightarrow{End}$ and $\overleftarrow{End}$, all the green circled cells are known in the forward computation. The information in $\overrightarrow{End}[7]$ can be backtracked to $SB^6[3]$, and a partial match constraint can be constructed as: $\texttt{SubBytes}(SB^6_F[3] \oplus SB^6_B[3]) = SB^6[3]$

Using $\zeta_{i,X'}$ and $\zeta_{i,Y'}$ to mark the eligible cells for X' and Y', the degree of match under MegaMC-Match is $m_k = \sum_{i=1}^{4} \zeta_{i,X'} + \zeta_{i,Y'} - 4$, since constructing the relation between X and Y will consume 4 DOFs. Note that to revert the calculations from Y' to Y and produce an eligible byte, the round key at the corresponding position must be 🟦 or ⬛.

4 Application to Rijndael

The preimage security of all versions of Rijndael-based hash functions is assessed with the refined and enhanced automation model. The Rijndael versions are indexed first by block size and then by key size, e.g. Rijndael 128-192 denotes the version of AES192. The results are given in Table 1.

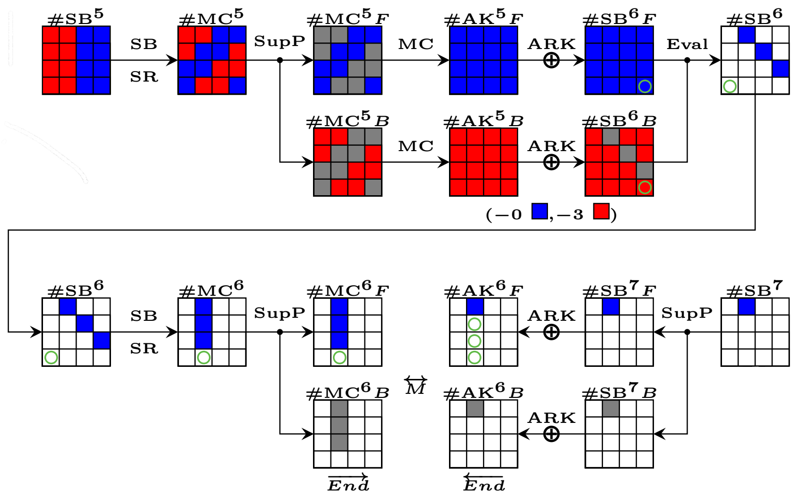


Fig. 5: Example of MegaMC-Match

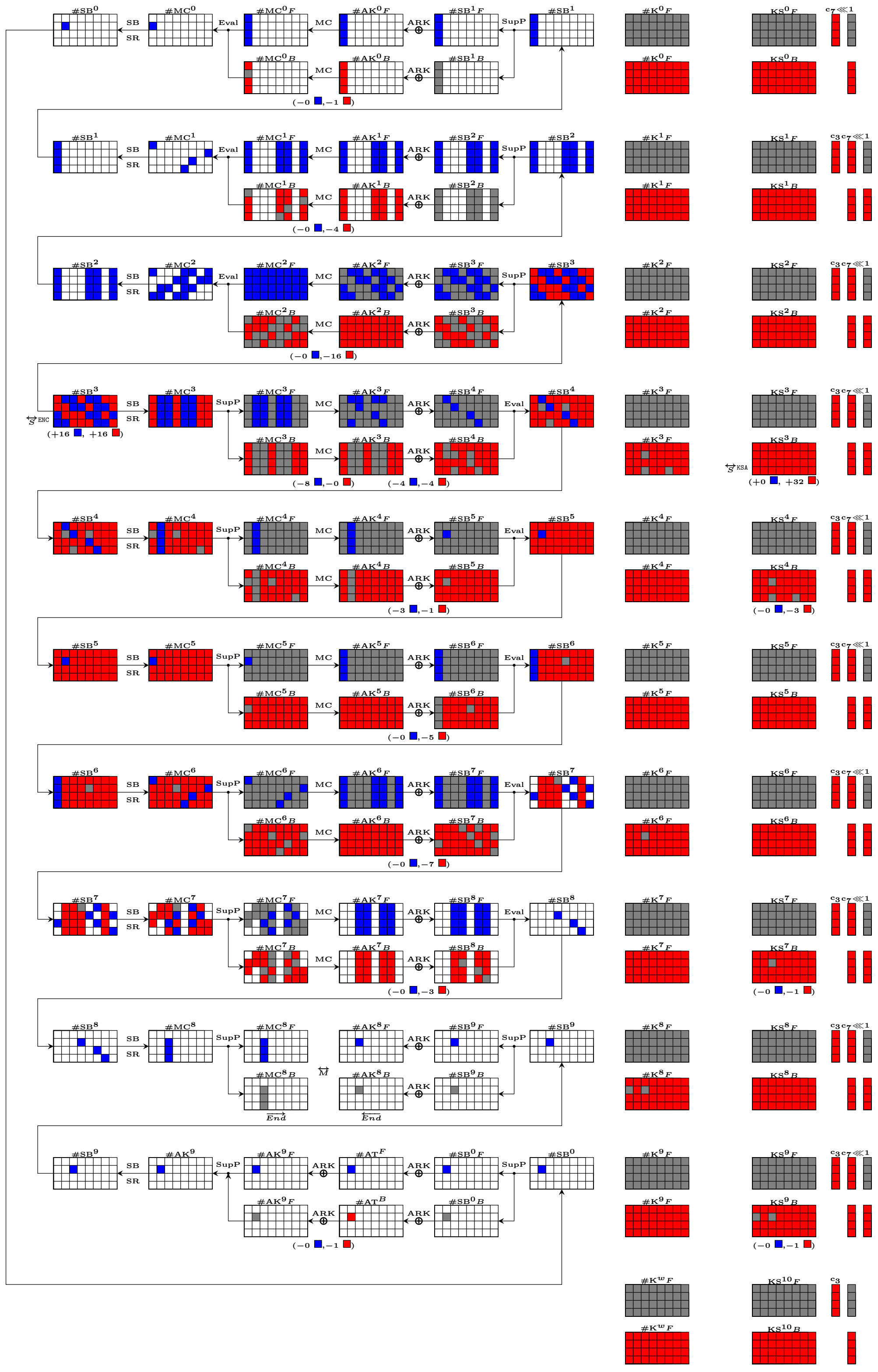
During the automatic search, the BiDir technique and the MegaMC-Match are critical techniques for better attack strategies. The Guess-and-Determine (GnD) strategy [8] and the Multiple AddRoundKey (MulAK) technique [8] are tested in the automatic search, both fail in yielding better results, either in attack rounds or complexity. The observation is in line with the declared Critical Tech. in Table 1 of [8].

In the figures, the intermediate states are indexed according to the type of operator to which they input, and the round index (i.e. SB^8 denotes the intermediate state immediately before the `SubBytes` operator in round 8). The special state AT denotes the intermediate state before XORing of a known text and the whitening key. Again, following traditional notation, a cell locates at the i-th row and the j-th column is indexed by $n = 4 \cdot j + i$.

4.1 Example: Pseudo-preimage Attack on 10-round Rijndael 256-256

The MITM attack procedures are demonstrated using Fig. 6, which depicts a pseudo-preimage attack on 10-round Rijndael 256-256. The attack starts with the precomputation of blue and red initial values. Recall that during propagation, certain cells are imposed constraints to preserve propagation trails, represented by the consumption of DOF and the coloring of ▨. A MITM attack fixes the value of such ▨ beforehand and precomputes the initial values satisfying those constraints.

Precomputation of red initial values. First, KS^4 and MC^3 are equivalently chosen as $\overleftrightarrow{S}^{\texttt{KSA}}$ and $\overleftrightarrow{S}^{\texttt{ENC}}$ in backward computation, i.e. $\mathcal{G}^{\texttt{KSA}} = \{9, 11, 23\}$,



Config

- $(\overrightarrow{c}, \overleftarrow{c})$ = (+16 🟦, +48 🟥) $\overleftrightarrow{S}^{\text{ENC}}$=(+16 🟦, +16 🟥) $\overleftrightarrow{S}^{\text{KSA}}$=(+0 🟦, +32 🟥)
- $(\overrightarrow{d_b}, \overleftarrow{d_r}, \overrightarrow{m}\overleftarrow{m}, \overrightarrow{d_{g_b}}, \overleftarrow{d_{g_r}}, \overleftrightarrow{d_{g_{br}}})$ = (+1 🟦, +1 🟥, +1 🟪, −0 🟦, −0 🟥, −0 🟩))

Fig. 6: A 10-round attack on Rijndael 256-256 with search objective 1.

$\mathcal{R}^{\texttt{KSA}} = \{0, 1, \ldots, 31\} \backslash \mathcal{G}^{\texttt{KSA}}$, $\mathcal{R}^{\texttt{ENC}} = \{0, 1, 2, 3, 12, 13, 14, 15, 24, 25, 26, 27, 28, 29, 30, 31\}$. All round keys can be expressed with free variables at $\mathcal{R}^{\texttt{KSA}}$ and predefined constants at $\mathcal{G}^{\texttt{KSA}} = \{kc_0^r, kc_1^r, kc_2^r\}$. There are two additional constraints imposed by ▩ at $KS^7[9] = kc_3^r$ and $KS^9[1] = kc_4^r$. A dependency in the key schedule appears at $KS^9[9]$: when $KS^7[9]$ and $KS^9[1]$ are ▩, $KS^9[9] = KS^8[9] \oplus KS^9[5] = KS^7[9] \oplus KS^9[1] = kc_3^r + kc_4^r$ is ▩ without DOF consumption. It is clear that the KeyXOR-Rule outperforms basic XOR-Rule since the position ▩ is essential for the forward propagation of $SB^9[9]$.

For encryption states, the active inputs on $\mathcal{R}^{\texttt{ENC}}$ can be constrained according to the predefined constants $c^r_{0,\ldots 41}$ locate at $SB^4_B[4, 5, 17, 18]$, $SB^5_B[5]$, $SB^6_B[0, 1, 2, 3, 17]$, $SB^7_B[2, 12, 17, 20, 22, 29, 31]$, $SB^8_B[13, 22, 27]$, $MC^0_B[1]$, $MC^1_B[0, 19, 22, 29]$, and $MC^2_B[0, 3, 6, 7, 10, 13, 15, 16, 17, 18, 19, 20, 22, 25, 28, 29]$.

To sum up, in backward computations, a total of $\overleftarrow{\sigma} = 47$ constraints have been added for $\overleftarrow{\iota} = 48$ variables according to the chosen values of predefined constants, leaving 2^8 valid candidates.

Precomputation of blue initial values. The precomputation of blue initial values is straightforward. $SB^5_F[5]$ can be selected equivalently to $\overleftrightarrow{S}^{\texttt{ENC}}$, and since are no constraints on $SB^5_F[5]$, all 2^8 candidates are valid. To simplify the forward propagation trail, the cost of DOF at MC^3 can be equivalently transferred to AK^3. Consequently, all cells in the forward computation can be expressed using the active byte $SB^5_F[5]$ and the predefined constants $c^b_{0,\ldots 14}$ located at $SB^5_F[4, 6, 7]$ and $SB^4_F[5, 6, 7, 8, 10, 11, 16, 17, 19, 20, 21, 22]$.

The pseudo-preimage attack procedure. The pseudo-preimage attack is performed as follows:

1. Select an untested set of predefined values for $c^r_{0,\ldots 41}$, $c^b_{0,\ldots 14}$, and $kc^r_{0,\ldots 4}$ from the pool of size 2^x, and initialize forward and backward lists L^f and L^B to empty.
2. Fix the symbolic gray cells $MC^5_F[0, 1, 2, 3, 8, 9, \ldots 31]$ as zeroes.
3. Feed the 2^8 candidates for forward computations into the computation path and compute to the matching point (i.e. $MC^8[9, 10, 11]$ and $AK^8[9]$).
4. With the 4 known blue cells at the matching point, calculate $MC^8[8] = AK^8[9] - 2MC^8[9] - 3MC^8[10] - MC^8[11]$ and subsequently $SB^8[8] = \texttt{SubBytes}(MC^8[8])$.
5. Store the candidates for forward computations into L^f, index by the value of $SB^8[8]$ and $SB^8_F[8]$.
6. Fix the symbolic gray cells $MC^3_B[4, 5, 6, 7, 8, 9, 10, 11, 16, 17, 18, 19, 20, 21, 22, 23]$ as zeros.

7. Feed the 2^8 candidates for backward computations into the computation path and compute to SB_B^8.
8. Store the candidates for backward computations into L^B, index by the value of $SB_B^8[8]$.
9. Check L^F and L^B for partial-match by testing if $SB^8[8]] = \texttt{SubBytes}(SB_F^8[8] \oplus SB_B^8[8])$. A total of 2^8 candidates is expected to remain in $L^F \times L^B$ ($2^8 = 2^8 \cdot 2^8 / \overrightarrow{\overleftarrow{m}}$).
10. Check the 2^8 candidates for a full match. If a full match is found, exit with the obtained pseudo-preimage of the given target. Otherwise, repeat procedures from 1 to 9.

Computational complexity. A remnant of 2^8 candidates (combined forward and backward) that satisfies the 8-bit partial-match can be obtained with one selection from the pool of 2^x potential predefined constant values. The value of x is calculated by $x = 256 - \overrightarrow{d_b} - \overleftarrow{d_r} = 240$. Thus, the time complexity of the above pseudo-preimage attack is $2^{x+8} = 2^{248}$. And by Equation 8, the time complexity of the converted preimage attack is calculated as $2^{(256+248)/2+1} = 2^{253}$

5 Conclusions

This work has further refined the MILP modeling of the automatic search of preimage attacks on the Rijndael structure. It has introduced a dedicated and lightweight encoding scheme for the superposition structure [8]. With new matching methods and treatment of dependencies in Rijndael's key schedule incorporated, this work has mounted the first comprehensive study on all versions of Rijndael-based hashing. This work has successfully replicated all referenced attacks and found improvements.

Further studies should focus on exploring and addressing more complex dependencies among neutral bytes to save repeated consumptions of DOFs, as well as discovering equivalence in attack patterns and pruning the search space.

References

1. Daemen, J., Rijmen, V.: AES proposal: Rijndael. In: NIST AES Proposal (1999).
2. Preneel, B., Govaerts, R., Vandewalle, J.: Hash functions based on block ciphers: A synthetic approach. In: CRYPTO 1993, pp. 368–378.
3. ZigBee Alliance: ZigBee Specification. ZigBee Document 053474r17 (2007). http://www.zigbee.org/
4. Aoki, K., Sasaki, Y.: Preimage attacks on one-block MD4, 63-step MD5 and more. In: SAC 2008, pp. 103–119.
5. Aumasson, J.-P., Meier, W., Mendel, F.: Preimage attacks on 3-pass HAVAL and step-reduced MD5. In: SAC 2008, pp. 120–135.

6. Bao, Z., Ding, L., Guo, J., Wang, H., Zhang, W.: Improved meet-in-the-middle preimage attacks against AES hashing modes. In: ToSC 2019, pp. 318–347.
7. Bao, Z., Dong, X., Guo, J., Li, Z., Shi, D., Sun, S., Wang, X.: Automatic search of meet-in-the-middle preimage attacks on AES-like hashing. In: EUROCRYPT 2021, Part I. pp. 771–804.
8. Bao, Z., Guo, J., Shi, D., Tu, Y.: Superposition meet-in-the-middle attacks: updates on fundamental security of AES-like hashing. In: CRYPTO 2022, Part I. pp. 64–93.
9. Barreto, P.S., Rijmen, V.: The Whirlpool hashing function. In: First open NESSIE Workshop, vol. 13, pp. 14. Citeseer
10. Bogdanov, A., Rechberger, C.: A 3-subset meet-in-the-middle attack: Cryptanalysis of the lightweight block cipher KTANTAN. In: SAC 2010, vol. 6544, pp. 229–240.
11. Bouillaguet, C., Derbez, P., Fouque, P.A.: Automatic search of attacks on round-reduced AES and applications. In: CRYPTO 2011, vol. 6841, pp. 169–187.
12. Dong, X., Hua, J., Sun, S., Li, Z., Wang, X., Hu, L.: Meet-in-the-middle attacks revisited: Key-recovery, collision, and preimage attacks. In: CRYPTO 2021, Part III, pp. 278–308.
13. Fuhr, T., Minaud, B.: Match box meet-in-the-middle attack against KATAN. In: FSE 2014, pp. 61–81.
14. Gauravaram, P., Knudsen, L.R., Matusiewicz, K., Mendel, F., Rechberger, C., Schläffer, M., Thomsen, S.S.: Grøstl a SHA-3 candidate. Retrieved from http://www.groestl.info/Groestl.pdf
15. Gilbert, H., Peyrin, T.: Super-sbox cryptanalysis: Improved attacks for AES-like permutations. In: FSE 2010, pp. 365–383.
16. Guo, J., Ling, S., Rechberger, C., Wang, H.: Advanced meet-in-the-middle preimage attacks: First results on full Tiger, and improved results on MD4 and SHA-2. In: ASIACRYPT 2010, pp. 56–75.
17. Sasaki, Y.: Meet-in-the-middle preimage attacks on AES hashing modes and an application to Whirlpool. In: FSE 2011, pp. 378–396.
18. Sasaki, Y., Aoki, K.: Preimage attacks on 3, 4, and 5-pass HAVAL. In: ASIACRYPT 2008, pp. 253–271.
19. Sasaki, Y., Aoki, K.: Finding preimages in full MD5 faster than exhaustive search. In: EUROCRYPT 2009, pp. 134–152.
20. Sasaki, Y., Aoki, K.. Preimage attacks on step-reduced MD5. In: ACISP 2008, pp. 282–296.
21. Aoki, K., Sasaki, Y.. Preimage attacks on one-block MD4, 63-step MD5 and more. In: SAC 2008, pp. 103–119.
22. Wang, L., Sasaki, Y., Wu, S., Wu, W.. Investigating fundamental security requirements on Whirlpool: Improved preimage and collision attacks. In: ASIACRYPT 2012, pp. 562–579.
23. Sun, S., Hu, L., Wang, P., Qiao, K., Ma, X., Song, L.. Automatic security evaluation and (related-key) differential characteristic search: Application to SIMON, PRESENT, LBlock, DES(L) and other bit-oriented block ciphers. In: ASIACRYPT 2014, pp. 158–178.
24. Wang, L., Sasaki, Y.. Finding preimages of Tiger up to 23 step. In: FSE 2010, pp. 116–133.
25. Wei, L., Rechberger, C., Guo, J., Wu, H., Wang, H., Ling, S.. Improved meet-in-the-middle cryptanalysis of KTANTAN. In: ACISP 2011, pp. 433–438.
26. Wu, S., Feng, D., Wu, W., Guo, J., Dong, L., Zou, J.. (Pseudo) preimage attack on round-reduced Grøstl hash function and others. In: FSE 2012, pp. 127–145.
27. Guo, J., Su, C., Yap, W.. An Improved Preimage Attack against HAVAL-3. *Inf. Process. Lett.* 115, no. 2 (2015), pp. 386–393.

28. Sasaki, Y., Wang, L., Wu, S., Wu, W.. Investigating fundamental security requirements on Whirlpool: Improved preimage and collision attacks. In: ASIACRYPT 2012, pp. 562–579.
29. Mouha, N., Wang, Q., Gu, D., Preneel, B.. Differential and linear cryptanalysis using mixed-integer linear programming. In: Inscrypt 2011, pp. 57–76.
30. Jean, J.. TikZ for Cryptographers, 2016. Available: https://www.iacr.org/authors/tikz/.
31. ISO/IEC. Information technology – Security techniques – Hash-functions – Part 2: Hash-functions using an n-bit block cipher.
32. Gauravaram, P., Knudsen, L., Matusiewicz, K., Mendel, F., Rechberger, C., Schl"affer, M., Thomsen, S.. Grøstl-a SHA-3 candidate. In: Dagstuhl Seminar Proceedings 2009.
33. Guo, J., Peyrin, T., Poschmann, A.. The PHOTON family of lightweight hash functions. In: CRYPTO 2011 Proceedings 31, pp. 222–239.
34. Guo, J., Peyrin, T., Poschmann, A., Robshaw, M. J. B. The LED Block Cipher. In: CHES 2011, pp. 326–341.
35. Diffie, W., Hellman, M. E. Special feature exhaustive cryptanalysis of the NBS data encryption standard. *Computer*, 10(6), 74–84, 1977.
36. Dunkelman, O., Sekar, G., Preneel, B. Improved meet-in-the-middle attacks on reduced-round DES. In: INDOCRYPT 2007 Proceedings 8, pp. 86–100.
37. Dong, X., Hua, J., Sun, S., Li, Z., Wang, X., Hu, L. Meet-in-the-middle attacks revisited: key-recovery, collision, and preimage attacks. In: CRYPTO 2021 Part III 41, pp. 278–308.
38. Wagner, D. The Boomerang Attack. In: FSE 1999.
39. Mendel, F., Rechberger, C., Schl"affer, M., Thomsen, S. S. The rebound attack: Cryptanalysis of reduced Whirlpool and Grøstl. In: FSE 2009, pp. 260–276.
40. Kocher, P. C. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: CRYPTO 1996 Proceedings 16, pp. 104–113.
41. Biham, E., Shamir, A. Differential cryptanalysis of DES-like cryptosystems. In: JOC vol. 4, pp. 3–72.
42. Matsui, M., Yamagishi, A. A new method for known plaintext attack of FEAL cipher. In: EUROCRYPT 1992 Proceedings 11, pp. 81–91.
43. Menezes, A. J., VanOorschot, P.C., Vanstone, S. A. *Handbook of applied cryptography* CRC Press Series on Discrete Mathematics and its Applications, 1997.
44. Leurent, G. MD4 is not one-way. In: FSE 2008 Revised Selected Papers 15, pp. 412–428.
45. Coron, J.-S., Dodis, Y., Malinaud, C., Puniya, P. Merkle-Damgård revisited: How to construct a hash function. In: CRYPTO 2005 Proceedings 25, vol. 3621, pp. 430-448.

Appendix

Table 2 and 3 show the number of iterations and `ShiftRows` offsets of Rijndael encryption.

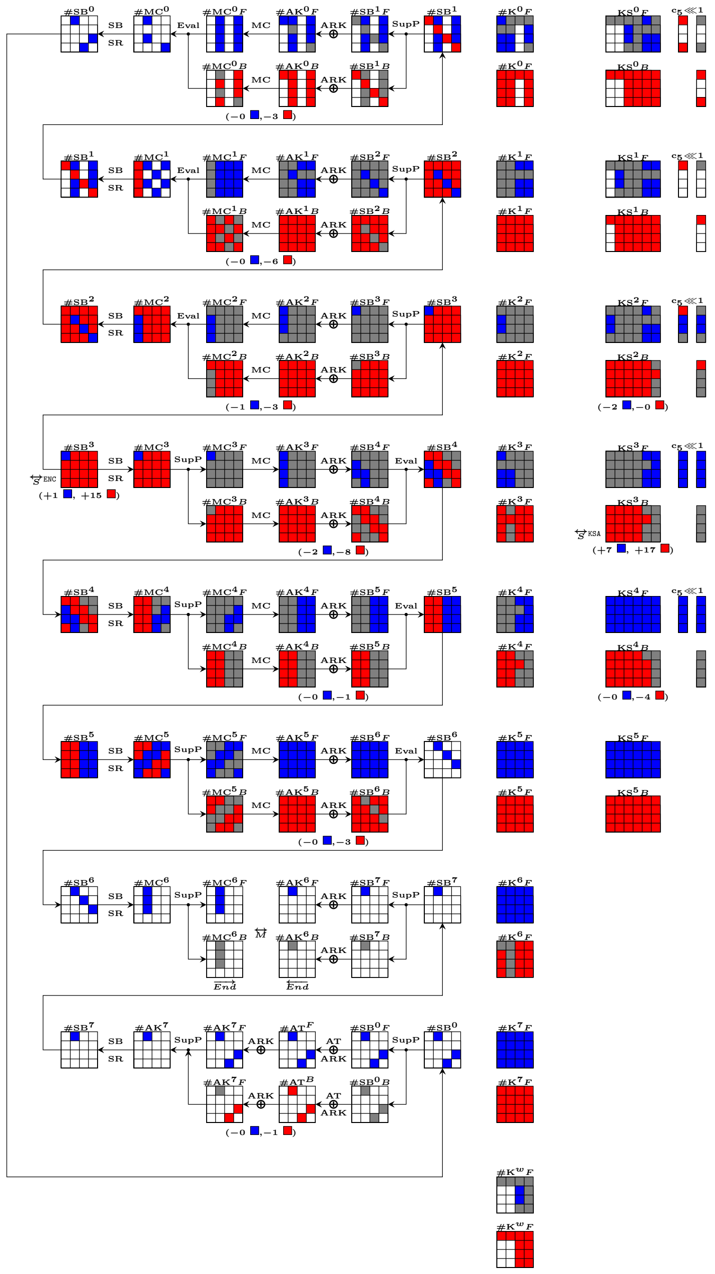
Table 2: Number of iteration

Nb	**Nk**	**Nr**
4	4	10
	6	12
	8	14
6	4	12
	6	12
	8	14
8	4	14
	6	14
	8	14

Table 3: `ShiftRows` offsets

Nb	**Row Index**	**Offset**
4/6	0	0
	1	1
	2	2
	3	3
8	0	0
	1	1
	2	3
	3	4

Fig. 7, 8, 9, 10, 11, and 12 show the attack figures for 8-round AES192, 9-round Rijndael 192-128, 192-192, 192-256, 256-128, and 10-round Rijndael 256-192.

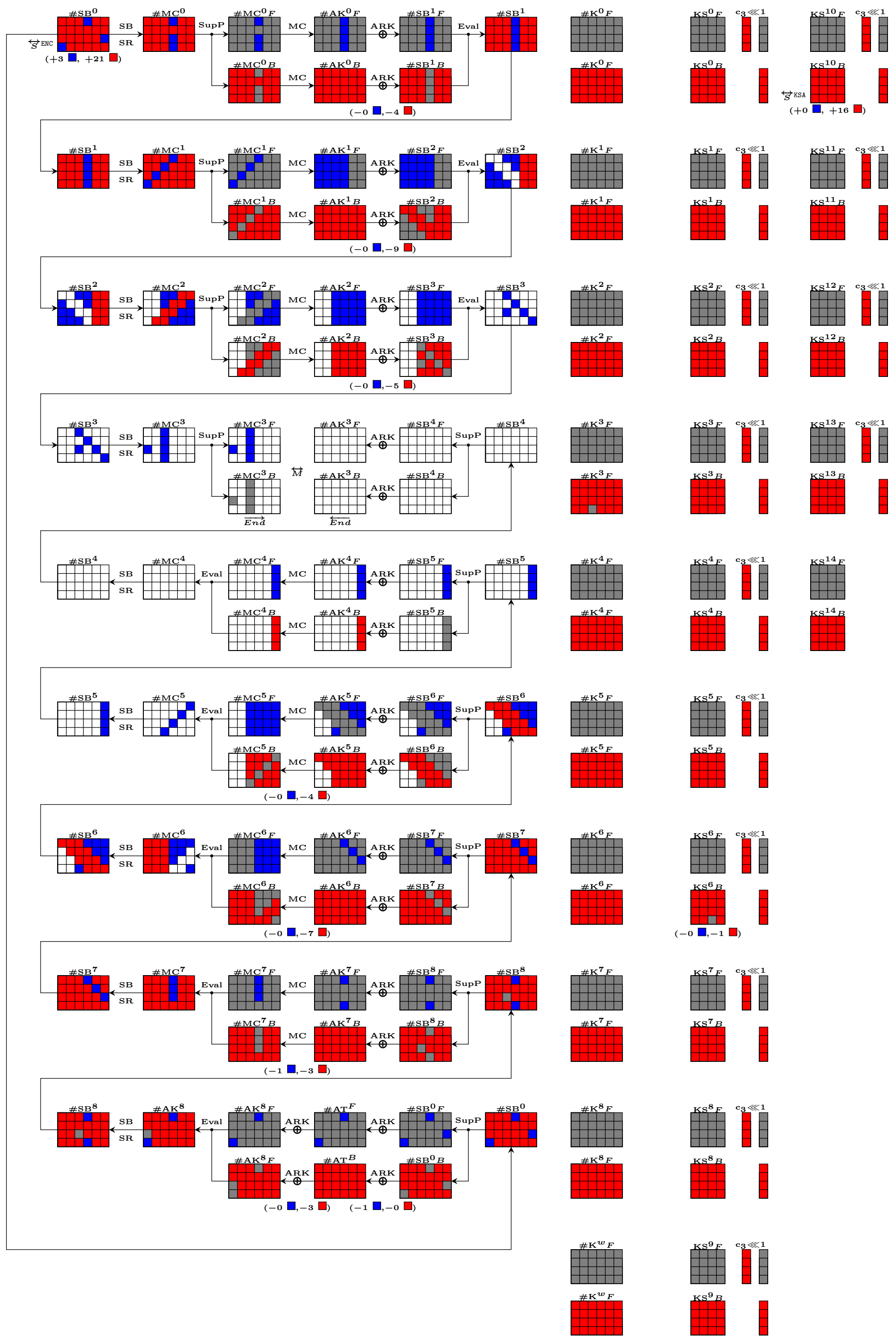


#SB⁰ SB SR #MC⁰ Eval #MC⁰F MC #AK⁰F ARK #SB¹F SupP #SB¹
#MC⁰B MC #AK⁰B ARK #SB¹B
(−0, −3)
#K⁰F KS⁰F c₅≪1
#K⁰F KS⁰B
#SB¹ SB SR #MC¹ Eval #MC¹F MC #AK¹F ARK #SB²F SupP #SB²
#MC¹B MC #AK¹B ARK #SB²B
(−0, −6)
#K¹F KS¹F c₅≪1 #K¹F KS¹B
#SB² SB SR #MC² Eval #MC²F MC #AK²F ARK #SB³F SupP #SB³
#MC²B MC #AK²B ARK #SB³B
(−1, −3)
#K²F KS²F c₅≪1 #K²F KS²B (−2, −0)
#SB³ SB SR #MC³ SupP #MC³F MC #AK³F ARK #SB⁴F Eval #SB⁴
S ENC (+1, +15)
#MC³B MC #AK³B ARK #SB⁴B
(−2, −8)
#K³F KS³F c₅≪1 #K³F KS³B S KSA (+7, +17)
#SB⁴ SB SR #MC⁴ SupP #MC⁴F MC #AK⁴F ARK #SB⁵F Eval #SB⁵
#MC⁴B MC #AK⁴B ARK #SB⁵B
(−0, −1)
#K⁴F KS⁴F c₅≪1 #K⁴F KS⁴B (−0, −4)
#SB⁵ SB SR #MC⁵ SupP #MC⁵F MC #AK⁵F ARK #SB⁶F Eval #SB⁶
#MC⁵B MC #AK⁵B ARK #SB⁶B
(−0, −3)
#K⁵F KS⁵F #K⁵F KS⁵B
#SB⁶ SB SR #MC⁶ SupP #MC⁶F #AK⁶F ARK #SB⁷F SupP #SB⁷
#MC⁶B M #AK⁶B ARK #SB⁷B
End End
#K⁶F #K⁶F
#SB⁷ SB SR #AK⁷ SupP #AK⁷F ARK #ATF AT ARK #SB⁰F SupP #SB⁰
#AK⁷F ARK #ATB AT ARK #SB⁰B
(−0, −1)
#K⁷F #K⁷F
#KʷF #KʷF

Config

- $(\overrightarrow{\iota}, \overleftarrow{\iota})$ = (+8 🟦, +32 🟥) $\overleftrightarrow{S}^{\text{ENC}}$=(+1 🟦, +15 🟥) $\overleftrightarrow{S}^{\text{KSA}}$=(+7 🟦, +17 🟥)
- $(\overrightarrow{d_b}, \overleftarrow{d_r}, \overrightarrow{m}\overleftarrow{}, \overrightarrow{d_{g_b}}, \overleftarrow{d_{g_r}}, \overleftrightarrow{d_{g_{br}}})$ = (+3 🟦, +3 🟥, +3 🟪, −0 🟦, −0 ▫, −0 🟩))

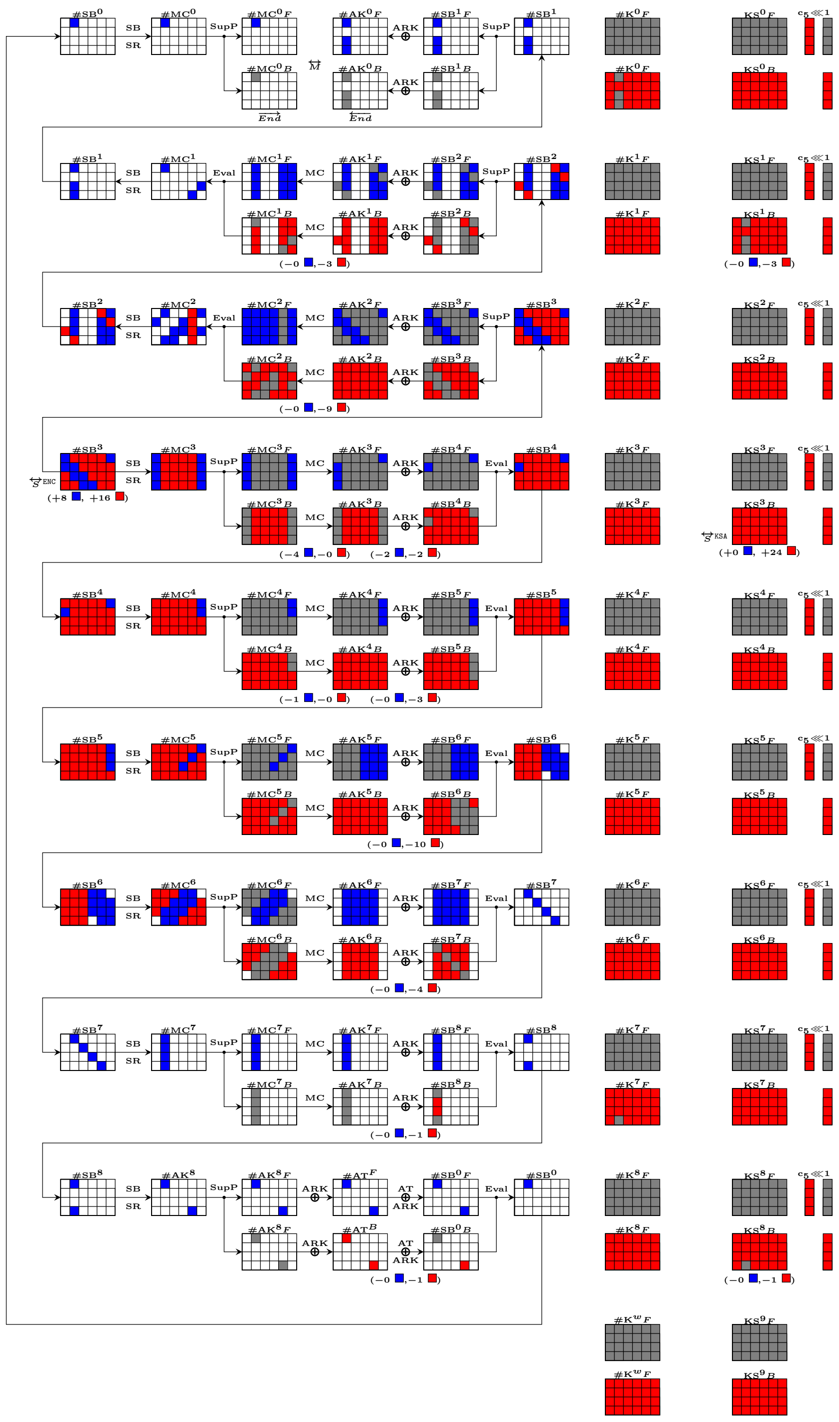
Fig. 7: A 8-round attack on Rijndael 128-192 with search objective 3.



Config

- $(\overrightarrow{\iota}, \overleftarrow{\iota})$ = (+3 🟦, +37 🟥) $\overleftrightarrow{S}^{\mathtt{ENC}}$=(+3 🟦, +21 🟥) $\overleftrightarrow{S}^{\mathtt{KSA}}$=(+0 🟦, +16 🟥)
- $(\overrightarrow{d_b}, \overleftarrow{d_r}, \overrightarrow{m}\overleftarrow{}, \overrightarrow{d_{g_b}}, \overleftarrow{d_{g_r}}, \overleftrightarrow{d_{g_{br}}})$ = (+1 🟦, +1 🟥, +1 🟪, −0 🟦, −0 🟥, −0 🟩))

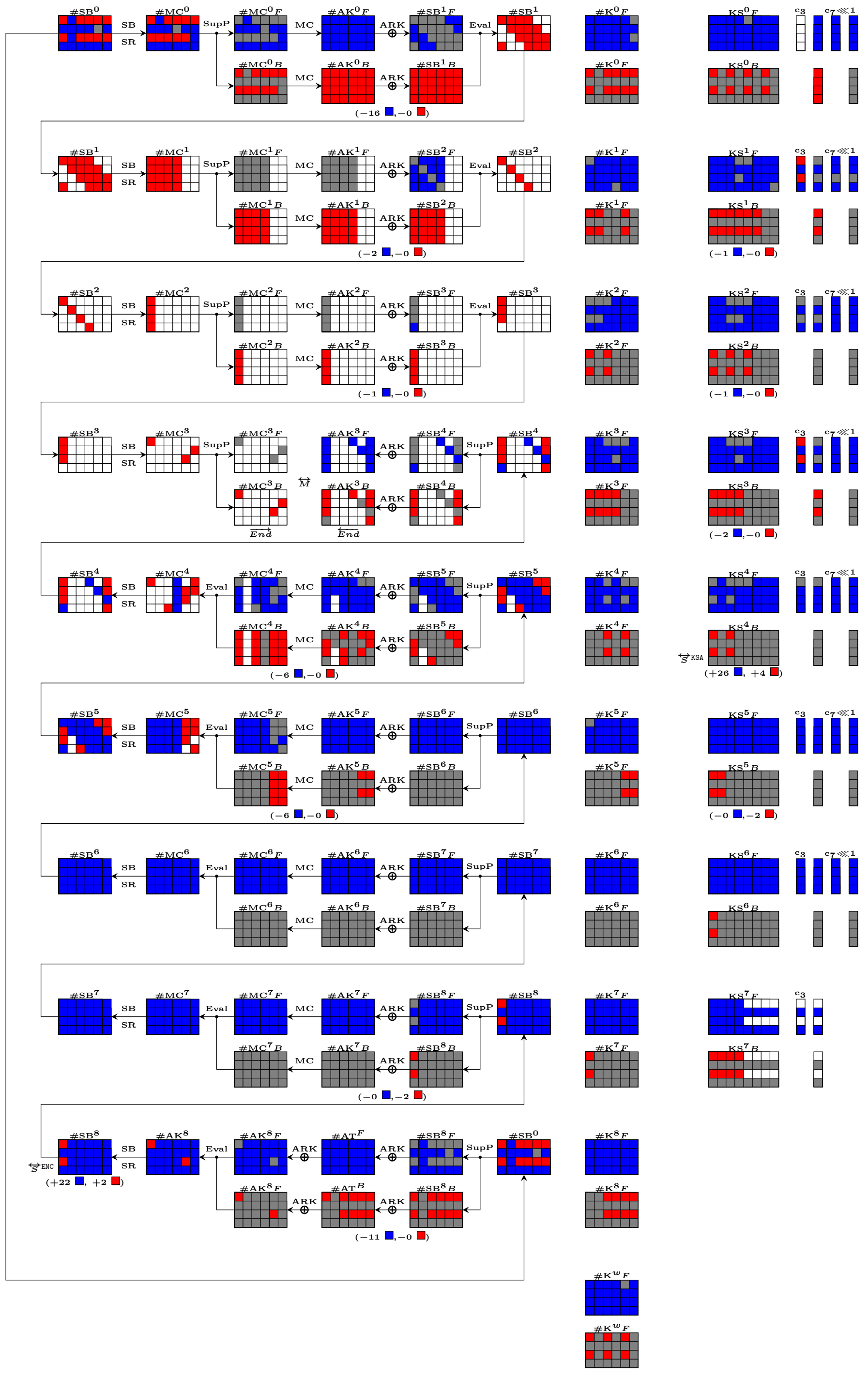
Fig. 8: An 8-round attack on Rijndael 192-128 with search objective 2.



Config

- ($\overrightarrow{\iota}$, $\overleftarrow{\iota}$) = (+8 🟦, +40 🟥) $\overleftrightarrow{S}^{\text{ENC}}$=(+8 🟦, +16 🟥) $\overleftrightarrow{S}^{\text{KSA}}$=(+0 🟦, +24 🟥)
- ($\overrightarrow{d_b}$, $\overleftarrow{d_r}$, $\overleftrightarrow{m}$, $\overrightarrow{d_{g_b}}$, $\overleftarrow{d_{g_r}}$, $\overleftrightarrow{d_{g_{br}}}$) = (+1 🟦, +3 🟥, +1 🟪, −0 🟦, −0 🟥, −0 🟩)

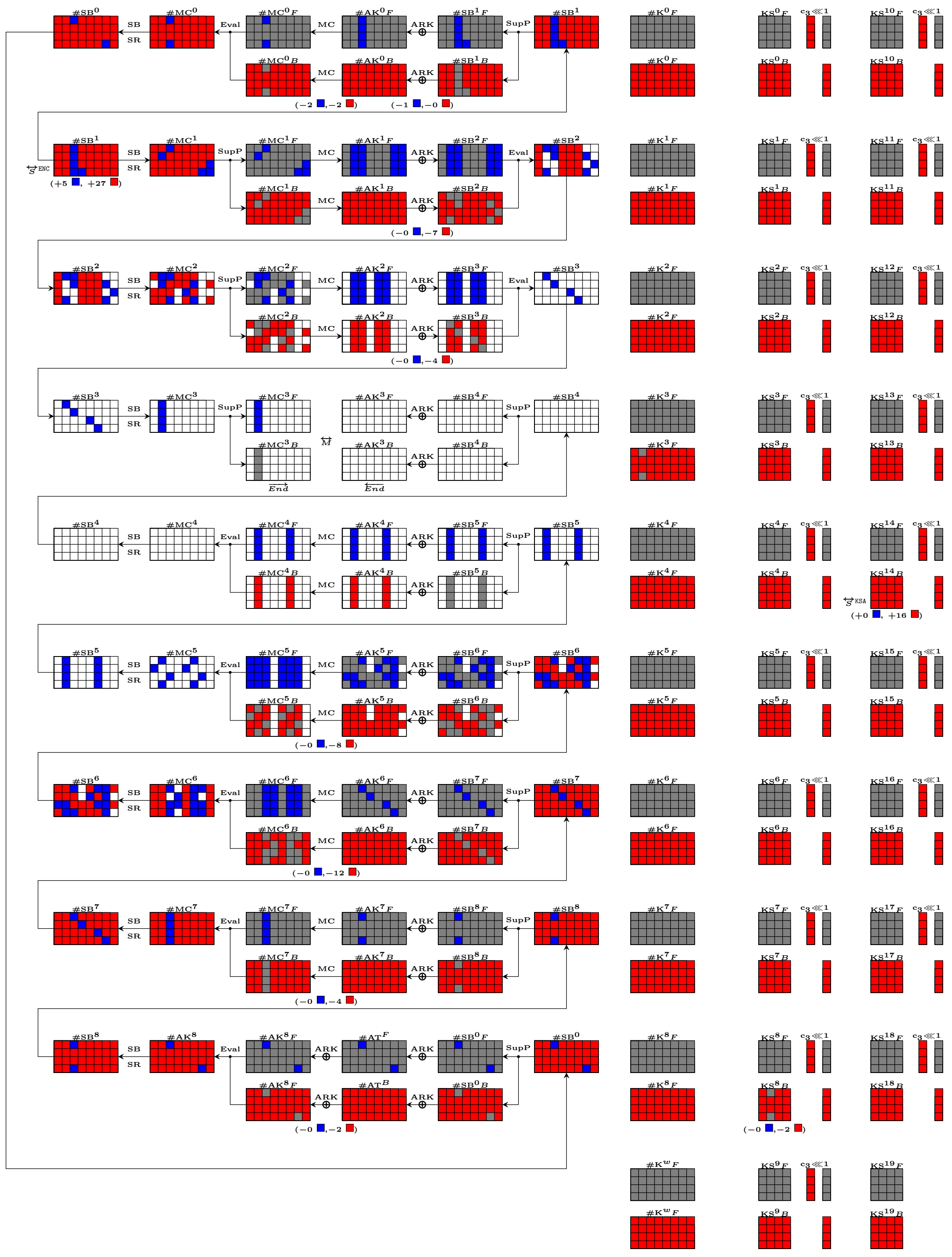
Fig. 9: An 9-round attack on Rijndael 192-192 with search objective 1.



Config

- $(\overrightarrow{\iota}, \overleftarrow{\iota}) = (+48$ 🟦, $+6$ 🟥$)$ $\overleftrightarrow{S}^{\texttt{ENC}} = (+22$ 🟦, $+2$ 🟥$)$ $\overleftrightarrow{S}^{\texttt{KSA}} = (+26$ 🟦, $+4$ 🟥$)$
- $(\overrightarrow{d_b}, \overleftarrow{d_r}, \overrightarrow{m}\overleftarrow{}, \overrightarrow{d_{g_b}}, \overleftarrow{d_{g_r}}, \overleftrightarrow{d_{g_{br}}}) = (+2$ 🟦, $+2$ 🟥, $+2$ 🟪, -0 🟦, -0 🟥, -0 🟩$)$

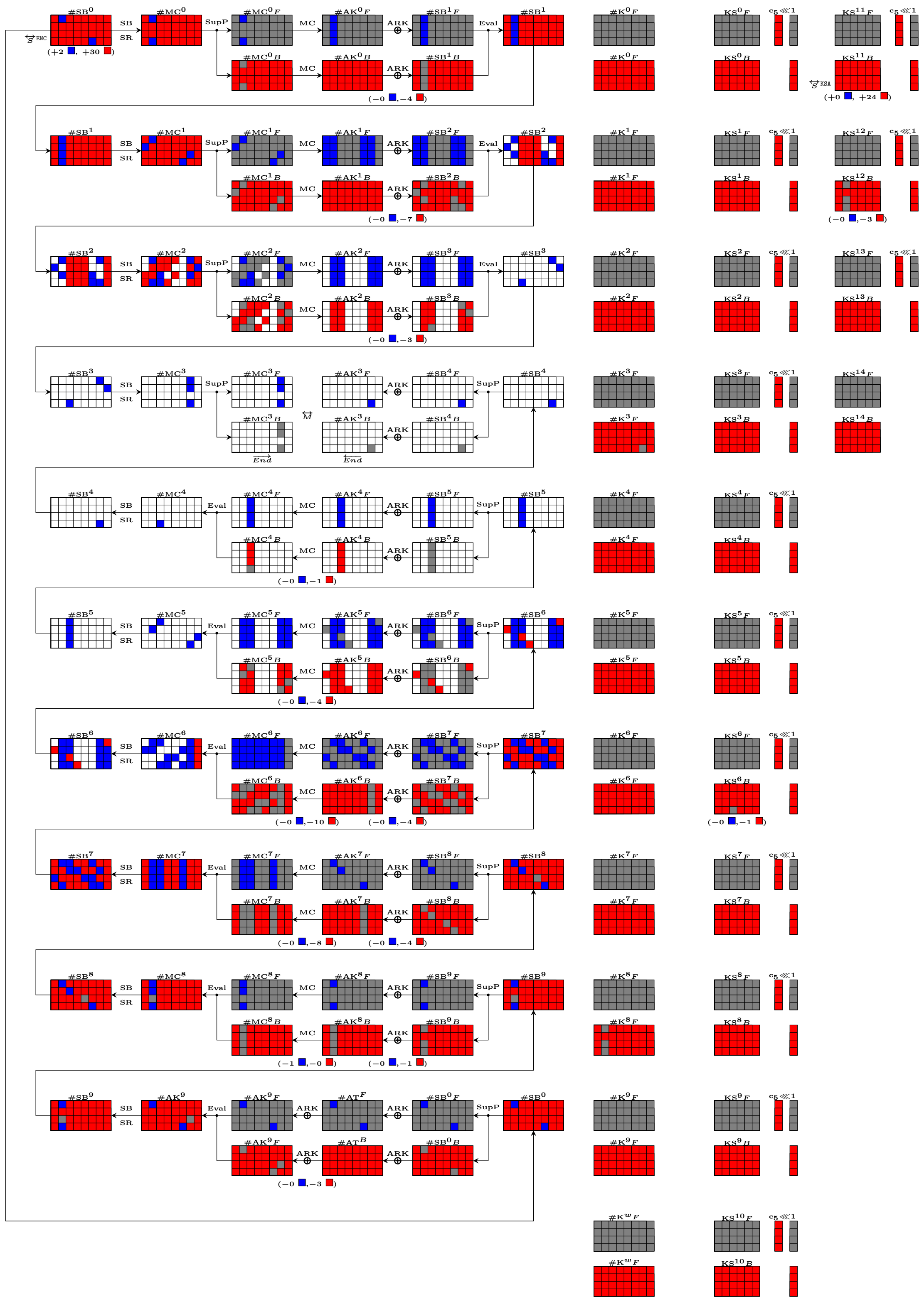
Fig. 10: A 9-round attack on Rijndael 192-256 with search objective 2.



Config

- $(\overrightarrow{\iota}, \overleftarrow{\iota}) = (+5$ 🟦, $+43$ 🟥) $\overleftrightarrow{S}^{\mathtt{ENC}} = (+5$ 🟦, $+27$ 🟥) $\overleftrightarrow{S}^{\mathtt{KSA}} = (+0$ 🟦, $+16$ 🟥)
- $(\overrightarrow{d_b}, \overleftarrow{d_r}, \overrightarrow{m}\overleftarrow{}, \overrightarrow{d_{g_b}}, \overleftarrow{d_{g_r}}, \overleftrightarrow{d_{g_{br}}}) = (+2$ 🟦, $+2$ 🟥, $+2$ 🟪, -0 🟦, -0 🟥, -0 🟩))

Fig. 11: An 9-round attack on Rijndael 256-128 with search objective 1.



Config

- $(\overrightarrow{\iota}, \overleftarrow{\iota})$ = (+2 🟦, +54 🟥) $\overleftrightarrow{S}^{\mathtt{ENC}}$=(+2 🟦, +30 🟥) $\overleftrightarrow{S}^{\mathtt{KSA}}$=(+0 🟦, +24 🟥)
- $(\overrightarrow{d_b}, \overleftarrow{d_r}, \overrightarrow{m}\overleftarrow{\ }, \overrightarrow{d_{g_b}}, \overleftarrow{d_{g_r}}, \overleftrightarrow{d_{g_{br}}})$ = (+1 🟦, +1 🟥, +1 🟪, −0 🟦, −0 🟥, −0 🟩))

Fig. 12: An 10-round attack on Rijndael 256-192 with search objective 1.