TWINE: A Lightweight, Versatile Block Cipher

Tomoyasu Suzaki, Kazuhiko Minematsu, Sumio Morioka, and Eita Kobayashi

NEC Corporation, 1753 Shimonumabe, Nakahara-Ku, Kawasaki, Japan
t-suzaki@cb.jp.nec.com, k-minematsu@ah.jp.nec.com,
s-morioka@ak.jp.nec.com, e-kobayashi@fg.jp.nec.com

Abstract. This paper presents a 64-bit lightweight block cipher TWINE supporting 80 and 128-bit keys. It enables quite small hardware implementation similar to the previous proposals, yet enables efficient implementations on embedded software. Moreover, it allows a compact implementation of unified encryption and decryption. This characteristic mainly originates from the use of generalized Feistel with many subblocks combined with a recent improvement on the diffusion layer.

Keywords: lightweight block cipher, generalized Feistel, block shuffle

1 Introduction

Recent advances in tiny computing devices, such as RFID and sensor network nodes, give rise to the need of symmetric encryption with highly-limited resources, called lightweight encryption. While we have AES it is often inappropriate for such devices due to their size/power/memory constraints, even though there are constant efforts for small-footprint AES, e.g., [14, 31, 37]. To fill the gap, a number of hardware-oriented lightweight block ciphers have been recently proposed; for instance, DESL [26], HIGHT [21], PRESENT [9], and KATAN/KTANTAN [13], PRINTcipher [25], and many more.

In this paper, we propose a new lightweight 64-bit block cipher TWINE. It supports 80 and 128-bit keys. Our purpose is to achieve hardware efficiency while minimizing the hardware-oriented design choices, such as a bit permutation. The avoidance of such options may be beneficial to software implementation and yield a balanced performance on both software and hardware. Lightweight block ciphers from a similar motivation are also seen in, e.g., KLEIN [19], LBlock [43], and most recently, LED [18] and Piccolo [40].

For this purpose, we employ Type-2 generalized Feistel structure (GFS) proposed\(^1\) by Zheng et al. [44] with 16 nibble-blocks. The drawback of such design is poor (slow) diffusion property, leading to a slow cipher with quite many rounds. To overcome the problem, we employ the idea of Suzuki and Minematsu at FSE ’10 [41] which substantially improves diffusion by using a different block shuffle from the original (cyclic shift). As a result, TWINE is also efficient on (embedded) software and enables compact unification of encryption and decryption.

\(^1\) Zheng et al. called it Type-2 Feistel-Type Transformation. The generalized Feistel structure is an alias taken by, e.g., [39, 41].
The features of our proposal are (1) no bit permutation, (2) generalized Feistel-based, and (3) no Galois-Field matrix. The components are only one 4-bit S-box and 4-bit XOR. As far as we know, this is the first attempt that combines these three features. There is a predecessor called LBlock [43] which has some resemblances to ours, however TWINE is an independent work and has several concrete design advantages (See Section 3).

We implemented TWINE on hardware (ASIC and FPGA) and software (8-bit microcontroller). We did not take the fixed key setting, hence keys can be updated. Our hardware implementations suggest that the encryption-only TWINE can be implemented with about 1,500 Gate Equivalent (GE), and when encryption and decryption are unified, it can be implemented within 1,800 GEs. We are also trying a serialized implementation and a preliminary result suggests 1,116 GEs. For software, TWINE is implemented within 0.8 to 1.5 Kbytes ROM.

The speed is relatively fast compared to other lightweight ciphers. Though the hardware size is not the smallest, we think the performance of TWINE is well-balanced for both hardware and software.

For security, TWINE employs a technique to enhance the diffusion of GFS, however, it is definitely important to evaluate the security against attacks that exploit the diffusion property of generalized Feistel, such as the impossible differential attack and the saturation attack. We perform a thorough analysis (for a new cipher proposal) on TWINE and present the impossible differential attack against 23-round TWINE-80 and 24-round TWINE-128 as the most powerful attacks we have found so far. The attack is ad-hoc and fully exploits the key schedule, which can be of independent interest as an example of highly-optimized impossible differential attack against GFS-based ciphers.

The organization of the paper is as follows. In Section 2 we describe the specification of TWINE. Section 3 explains the design rationale for TWINE. In Section 4 we present the result of security evaluation, and in Section 5 we present the implementation results of both hardware and software. Section 6 concludes the paper.

2 Specification of TWINE

2.1 Notations

The basic mathematical operations we use are as follows. ⊕ denotes bitwise exclusive-OR. For binary strings, x and y, x∥y denotes the concatenation of x and y. Let |x| denote the bit length of a binary string x. If |x| = m, x is also written as x(m) to emphasize its bit length. If |x| = 4c for some positive integer c, we write x → (x0∥x1∥...∥x_{c-1}), where |x_i| = 4, is the partition operation into the 4-bit subsequences. The opposite operation, (x0∥x1∥...∥x_{c-1}) → x, is similarly defined. The partition operation may be implicit, i.e., we may simply write x_i to denote the i-th 4-bit subsequence for any 4c-bit string x.
2.2 Data Processing Part (Encryption Process)

TWINE is a 64-bit block cipher with two supported key lengths, 80 and 128 bits. If the key length is needed to be specified, we write TWINE-80 or TWINE-128 to denote the corresponding version. The global structure of TWINE is a variant of Type-2 GFS \cite{44} \cite{38} with 16 4-bit sub-blocks. Given a 64-bit plaintext, \( P^{(64)} \), and a round key, \( RK^{(32 \times 36)} \), the cipher produces the ciphertext \( C^{(64)} \). Round key \( RK^{(32 \times 36)} \) is derived from the secret key, \( K^{(n)} \) with \( n \in \{80, 128\} \), using the key schedule. A round function of TWINE consists of a nonlinear layer using 4-bit S-boxes and a diffusion layer, which permutes the 16 blocks. Unlike Type-2 GFS, the diffusion layer is not a circular shift and is designed to provide a better diffusion that the circular shift, according the result of \cite{41}. This round function is iterated for 36 times for both key lengths, where the diffusion layer of the last round is omitted. The encryption process can be written as Algorithm 2.1.

The S-box, \( S \), is a 4-bit permutation defined as Table 1. The permutation of block indexes, \( \pi : \{0, \ldots, 15\} \rightarrow \{0, \ldots, 15\} \), where \( j \)-th sub-block (for \( j = 0, \ldots, 15 \)) is mapped to \( \pi[j] \)-th sub-block, is depicted at Table 2.

The figure of the round function is in Fig. 1.

Algorithm 2.1: TWINE.Enc\((P^{(64)}, RK^{(32 \times 36)}, C^{(64)})\)

\[
X_{i}^{(64)} \leftarrow P \\
RK_{0}^{(32)} \parallel \ldots \parallel RK_{35}^{(32)} \leftarrow RK_{32 \times 36} \\
\text{for } i \leftarrow 1 \text{ to } 35 \text{ do}
\begin{align*}
&X_{0(i)}^{(4)} \parallel X_{1(i)}^{(4)} \parallel \ldots \parallel X_{14(i)}^{(4)} \parallel X_{15(i)}^{(4)} \leftarrow X_{i}^{(64)} \\
&RK_{0(i)}^{(4)} \parallel RK_{1(i)}^{(4)} \parallel \ldots \parallel RK_{6(i)}^{(4)} \parallel RK_{7(i)}^{(4)} \leftarrow RK_{i}^{(32)} \\
&\text{for } j \leftarrow 0 \text{ to } 7 \text{ do}
\begin{align*}
&\text{do } X_{2j+1}^{i+1} \leftarrow S(X_{2j}^{i} \oplus RK_{j}^{i}) \oplus X_{2j+1}^{i} \\
&\text{for } h \leftarrow 0 \text{ to } 15 \text{ do}
\begin{align*}
&\text{do } X_{i+1}^{h} \leftarrow X_{h}^{i} \\
&X_{i+1}^{h} \leftarrow X_{i+1}^{h} \parallel X_{i+1}^{h+1} \parallel \ldots \parallel X_{i+1}^{h} \parallel X_{i+1}^{h+1} \\
&\text{for } j \leftarrow 0 \text{ to } 7 \text{ do}
\begin{align*}
&\text{do } X_{36j+1}^{36} \leftarrow S(X_{36j}^{36} \oplus RK_{j}^{36}) \oplus X_{36j+1}^{36} \\
&C \leftarrow X_{36}^{36}
\end{align*}
\end{align*}
\end{align*}
\end{align*}
\]

Table 1. S-box Mapping in the Hexadecimal Notation.

<table>
<thead>
<tr>
<th>( x )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S(x) )</td>
<td>C</td>
<td>0</td>
<td>F</td>
<td>A</td>
<td>2</td>
<td>B</td>
<td>9</td>
<td>5</td>
<td>8</td>
<td>3</td>
<td>D</td>
<td>7</td>
<td>1</td>
<td>E</td>
<td>6</td>
<td>4</td>
</tr>
</tbody>
</table>
Fig. 1. Round function of TWINE. Data path is 4-bit and each F function is of the form \( y = S(x \oplus k) \) with a 4-bit S-box.

<table>
<thead>
<tr>
<th>( \pi[j] )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \pi^{-1}[j] )</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>7</td>
<td>12</td>
<td>3</td>
<td>8</td>
<td>13</td>
<td>6</td>
<td>9</td>
<td>2</td>
<td>15</td>
<td>10</td>
<td>11</td>
<td>14</td>
</tr>
</tbody>
</table>

### Table 2. Block Shuffle \( \pi \) and its Inverse \( \pi^{-1} \).

2.3 Key Schedule Part

The key schedule produces \( RK_{(32 \times 36)} \) from the secret key, \( K(n) \), where \( n \in \{80, 128\} \). As well as the data processing part, it is a variant of GFS but with much sparser nonlinear functions. The pseudocode of key schedule for 80-bit key is in Algorithm 2.2 and the figure is in Appendix C. For 128-bit key, see Appendix A. Round constant, \( \text{CON}_i \), is defined as \( 2^i \) in \( \text{GF}(2^{6}) \) with primitive polynomial \( z^{6} + z + 1 \). The exact values are listed at Table 3.

**Table 3. Round Constants.** \( \text{CON}_i \) is the rightmost 6-bit (of 8 bits expressed in hexadecimal notation).

| \( i \) | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| \( \text{CON}_i \) | 01 | 02 | 04 | 08 | 10 | 20 | 03 | 06 | 0C | 18 | 30 | 23 | 05 | 0A | 14 | 28 | 13 | 26 |

| \( i \) | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 |
| \( \text{CON}_i \) | 0E | 18 | 3C | 38 | 35 | 29 | 11 | 22 | 07 | 0E | 1C | 38 | 33 | 25 | 09 | 12 | 24 | 08 |

4
2.4 Decryption Process

The decryption of TWINE is quite similar to the encryption; we use the same S-box and key schedule as used in the encryption, with the inverse block shuffle. See Algorithm 2.3.

Algorithm 2.3: TWINE.Dec($C(64)$, $RK_{(32 \times 36)}$, $P_{(64)}$)

\[
\begin{align*}
X_{(64)} & \leftarrow C \\
RK_{0(32)}^{j} & \| \cdots \| RK_{2(32)}^{j} - RK_{(32 \times 36)} \\
\text{for } i = 36 \text{ to } 2 & \text{ do } \begin{cases} 
X_{(64)}^{i} & \| \cdots \| X_{15(4)}^{i} \leftarrow X_{(64)}^{i} \\
RK_{0(4)}^{j} & \| RK_{2(4)}^{j} \| \cdots \| RK_{15(4)}^{j} \leftarrow RK_{(32)}^{j} \\
\text{for } j = 0 \text{ to } 7 & \text{ do } \begin{cases} 
X_{2j+1}^{i} & \leftarrow S(X_{2j}^{i} \oplus RK_{2}^{i}) + X_{2j+1}^{i} \\
\text{for } h = 0 \text{ to } 15 & \text{do } X_{h}^{i-1} \leftarrow X_{h}^{i} \\
X_{1}^{i-1} & \leftarrow X_{6}^{i-1} \| X_{1}^{i-1} \| \cdots \| X_{15(4)}^{i-1} \leftarrow X_{15}^{i-1} \\
\text{for } j = 0 \text{ to } 7 & \text{do } X_{2j+1}^{i} \leftarrow S(X_{2j}^{i} \oplus RK_{2}^{i}) + X_{2j+1}^{i} \\
P & \leftarrow X_{1}^{i}
\end{cases}
\end{cases}
\end{align*}
\]
3 Design Rationale

3.1 Basic Objective

We focus on the mixed environments of resource-constrained hardware and software, and aim at building a block cipher with a balanced performance under such environments. Specifically, our goals are

1. small footprint in hardware implementation (e.g. under 2,000 GE [23,33]),
2. small ROM/RAM consumption in software implementation,
3. these goals achieved for the unified encryption/decryption functionality.

The importance of the last item is also pointed out by [1].

On LBlock. We remark that LBlock [43], proposed independently of ours, is quite similar to our proposal. It is a 64-bit block cipher based on the balanced Feistel whose round function consists of 8 4-bit S-boxes followed by a 4-bit block-wise permutation (hence no matrix operation as used by SPN). LBlock also performs 8-bit rotation to the round function’s output. Such a structure can be transformed into a further generalized Type-2 GFS proposed by [41], though we do not know whether the authors of LBlock are aware of it. We investigated LBlock in this respect and found that the LBlock’s diffusion layer is equivalent to that of the decryption of our proposal. Note that this choice is quite reasonable from Table 6 of [41], as it satisfies both of the fastest diffusion and the highest immunities against linear and differential attacks among other block shuffles.

Nevertheless, there are some important differences between TWINE and LBlock, as follows;

1. LBlock uses ten distinct S-boxes while ours uses one S-box. The use of single S-box rather than multiple ones can contribute to smaller (serialized) hardware and software implementations.
2. LBlock uses a bit permutation in its key scheduling, while ours is completely bit permutation-free, including the key schedule. Hence the design of LBlock does not meet our criteria mentioned at Introduction.

We also would like to point out that the security evaluation of LBlock is insufficient. We already found a saturation attack against 22-round version without considering the key schedule, thus the security margin is smaller than the claimed by the authors (20-round). Using the techniques presented at Section 4, we expect further improvements on the attack.

3.2 Parameters and Components

Considering the basic design goals as above, we choose the 64-bit block size with 80 and 128-bit keys, which is compatible to many previous lightweight blockciphers. The number of rounds is determined from our security analysis. As far as we investigated, the most powerful attack against TWINE is a dedicated
impossible differential attack, which breaks 23-round TWINE-80 and 24-round TWINE-128. From this, we consider 36-round TWINE-128 has a sufficient security margin. When keeping the same size of margin for TWINE-80 and TWINE-128 is our sole goal, the 80-bit key version could reduce the number of rounds from 36. However, considering the security margin and the implementation merit (i.e. 36 has many factors, which enables various multiple-round hardware implementations with a small overhead), we employ the 36-round structure for both key lengths.

Block Shuffle. The block shuffle $\pi$ comes from a result of FSE ’10 [41]. In [41], it was reported that by changing the block shuffle different from the ordinal cyclic shift one can greatly improve the diffusion of Type-2 GFS. Here, goodness-of-diffusion is measured by the minimum number of rounds thatdiffuses any input sub-block difference to all output sub-blocks, called DRmax. Smaller DRmax means a faster diffusion. DRmax of cyclic shift with $k$ sub-blocks is $k$, while there exist shuffles with DRmax = $2 \log_2 k$, called “optimum block shuffle” [41]. Our $\pi$ is such one\(^2\) with $k = 16$, hence DRmax = 8 while DRmax = 16 for the cyclic shift. DRmax is connected to the resistance against various attacks. For example, Type-2 GFS with 16 sub-blocks has 33-round impossible differential characteristics and 32-round saturation characteristics. However, by using $\pi$ of Table 2 they can be reduced to 14 and 15 rounds.

There exist multiple optimum block shuffles [41]. Hence $\pi$ was chosen considering other aspects which is not (directly) related to DRmax. In particular, we focus on the resistance against differential and linear cryptanalysis, i.e., the number of active S-boxes.

S-box. The 4-bit S-box is chosen to satisfy
1. The maximum differential and linear probabilities are $2^{-2}$, which is theoretically the minimum for invertible S-box,
2. The boolean degree is 3,
3. The interpolation polynomial contains many terms and has degree 14.

Following the AES S-box design, we searched S-boxes satisfying the above while being representable as a composition of Galois field inversion and an affine transformation. More precisely, our S-box is defined as $y = S(x) = f((x \oplus b)^{-1})$, where $a^{-1}$ denotes the inverse of $a$ in GF($2^4$) (the zero element is mapped to itself) with irreducible polynomial $z^4 + z + 1$, and $b = 1$ is a constant, and $f(\cdot)$ is an affine function defined as

$$
\begin{bmatrix}
  y_0 \\
  y_1 \\
  y_2 \\
  y_3
\end{bmatrix} =
\begin{bmatrix}
  0 & 0 & 1 & 1 \\
  1 & 0 & 0 & 1 \\
  1 & 0 & 0 & 0 \\
  0 & 1 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
  x_0 \\
  x_1 \\
  x_2 \\
  x_3
\end{bmatrix}
$$

(1)

for $y = f(x)$ with $y = (y_0 \| y_1 \| y_2 \| y_3)$ and $x = (x_0 \| x_1 \| x_2 \| x_3)$.

\(^2\) More precisely, an isomorphic shuffle in Appendix B ($k = 16$, No. 10) of [41].
Key Schedule. The key schedule has many design options. We choose one which enables (1) on-the-fly operations and (2) produces each round key via sequential update of a key state, that is, there is no intermediate key. As mentioned, it uses no bit permutation. As hardware efficiency is not our ultimate goal, the design is rather conservative compared to the recent hardware-oriented ones [13,34,40], yet quite simple. For security, we want our key schedule to have sufficient resistance against slide, meet-in-the-middle, and related-key attacks.

4 Security Evaluation

4.1 Overview

We examined the security of TWINE against various attacks for both 80 and 128-bit keys. Since it is hard to describe all the evaluation results due to the page limit, we focus on the most critical attacks in our evaluation; the impossible differential and saturation attacks. For simplicity we only describe the details of the attacks against TWINE-80; the results on TWINE-128 will be briefly described in the summary. The short summary on other attacks, such as differential and linear attacks, will also be given.

In this section, we use the following notations. \( \bar{S}^c \) denotes the sequence of \( c \) symbols \( S \), e.g. \( 0^3 \) means \((0,0,0)\) and \( U^3 \) means \( (U,U,U) \). The \( F \) function in the \( i \)-th round is labeled as \( F^i_0, \ldots, F^i_{15} \), where \( F^i_0 \) is the leftmost one. We let \( RK^i_{j_1, \ldots, j_h} \) to denote the vector \((RK^i_{j_1}, \ldots, RK^i_{j_h})\). Since \( RK^i_{j} \) is the \( j \)-th 4-bit subsequence of \( RK^i \) (for \( j = 0,1, \ldots, 15 \)), this means \( F^i_j(x) = S(RK^i_j \oplus x) \). \( X^i_{j_1, \ldots, j_h} \) is similarly defined.

4.2 Impossible Differential Attack

Generally, impossible differential attack [6] is one of the most powerful attack against Feistel and GFS-based ciphers, as demonstrated by (e.g.) [15, 32, 42]. We searched impossible differential characteristics (IDCs) using Kim et al.’s method [24], and found 64 14-round IDCs. We here present a highly-optimized attack against 23-round TWINE-80. It exploits the key schedule and is based on the following 14-round IDC (for 4-bit blocks);

\[
(0, \alpha, \bar{0}^{14})^{14r}(\bar{0}^{8}, \beta, \bar{0}^7), \quad \text{where } \alpha \neq 0 \text{ and } \beta \neq 0.
\]

Our attack uses the above IDC to 5-th to 18-th rounds of TWINE, and tries to recover the subkeys of the first 4 rounds and last 5 rounds, 144 bits in total. These subkey bits are uniquely determined via its 80-bit subsequence; see Appendix D.

The details of our attack are as follows.

Data Collection. We call a set of \( 2^{32} \) plaintexts a structure if its \( i \)-th sub-blocks are fixed to a constant for all \( i = 2,4,5,6,7,8,9,14 \in \{0, \ldots, 15\} \) and the remaining 8 sub-blocks take all \( 2^{32} \) values. Suppose we have one structure. From it we extract plaintext pairs having the differential

\[
(\alpha_1, \alpha_2, 0, \alpha_3, \bar{0}^6, \alpha_4, \alpha_5, \alpha_6, \alpha_7, 0, \alpha_0),
\]

\( (\alpha_1, \alpha_2, 0, \alpha_3, \bar{0}^6, \alpha_4, \alpha_5, \alpha_6, \alpha_7, 0, \alpha_0) \),
where $\alpha_1$ is a non-zero 4-bit value. For such a plaintext pair, we want to make sure that the differential of the internal state after the first 4 rounds to match the left hand side of Eq. (2). For this, the output differentials with respect to some $F$ functions (in the first 4 rounds) have to be canceled out. For example, $\alpha_2$ must be the differential of $F$ with input differential $\alpha_1$, as shown by Fig. 2.

Here, we use the following observation:

**Proposition 1.** Let $y = F^j_i(x) = S(RK^j_i \oplus x)$ and $y' = F^j_i(x')$. For fixed $\Delta_x = x \oplus x' \neq 0$, $\Delta_y = y \oplus y'$ has always 7 possible values, for any $i$ and $j$. Moreover, for a fixed $\Delta_x \neq 0$ let $\tau(\Delta_y)$ be the function of $\Delta_y$ which represents the number of possible $RK^j_i$ values. Then $\tau(\Delta_y)$ equals 2 for some 6 values of $\Delta_y$ and 4 for the remaining one.

Hence we have a set of 7 possible values for $\alpha_2$, which is determined by $\alpha_1$. Considering this restriction, we can extract $2^{54.56}$ plaintext pairs from a structure with its differential being Eq. (3).

**Key Elimination.** After the plaintext pairs have been generated, we encrypt them and seek the ciphertext pairs with a differential

$$(0, \beta_1, 0, \beta_2, \beta_3, \beta_4, \beta_0, \beta_5, \beta_7, \beta_8, \beta_9, \beta_{10}, \beta_{11}, 0, 0), \forall \beta_i \neq 0. \tag{4}$$

For each ciphertext pair with differential Eq. (4), we try to eliminate the wrong (impossible) candidates for the 80-bit (sub)key vector $(K_1||K_2||K_3)$, where

$$K_1 = (RK^{1}_{[1,2,3,7]}, RK^{23}_{6}), K_2 = (RK^{1}_{[0,5,6]}, RK^{2}_{[2,4,6,7]}, RK^{23}_{[2,4,6]}), RK^{22}_{[1,3,4]}),$$
$$K_3 = (RK^{22}_{[0,2]}), \tag{5}$$

using the plaintext pair of differential Eq. (3) and the ciphertext pair of differential Eq. (4). This can be done as follows. First, we guess the 20-bit $K_1$ (which can take all possible values). After $K_1$ is guessed, the number of each 4-bit subkey candidates in $K_2$ is $(2 \cdot 6 + 4)/7 \approx 2.28$ on average from Proposition 1. Moreover, once $K_1$ and $K_2$ are fixed, each 4-bit subkey of $K_3$ will have $(2 \cdot 6 + 4)/15 \approx 1.07$ candidates, as we have no restrictions on the input differential for $F$’s relating to these subkeys. From this observation, we can expect to eliminate $2^{20}, (16/7)^{13}, (16/15)^2 \approx 2^{15.69}$ candidates from a set of $2^{90}$ values for each plaintext-ciphertext pair, i.e. a guess of 80-bit $(K_1||K_2||K_3)$ is eliminated with probability $2^{-44.31}$. The detail of the above procedure is depicted at Table 12 in Appendix D.

From this, to determine $(K_1||K_2||K_3)$ with probability almost one, we need $N$ ciphertext pairs, where $N$ satisfies $2^{90}(1 - 2^{-44.31})^N \approx 1$. This implies $N \approx 2^{50.11}$. Assuming the ciphertext’s randomness, we can expect a ciphertext pair of differential Eq. (4) with probability $(2^{-4})^4 \cdot (2^{-1})^8 = 2^{-24}$. This implies that we basically need $2^{74.11}$ ciphertext pairs. But in fact we need some more. In the key elimination we need to compute some other subkeys (64 bits in total), which is uniquely determined by the key of Eq. (5). These keys contain $RK^4_1, RK^4_2$, and $RK^6_6$ and they can cause a contradiction with other keys. If this event occurs,
the corresponding plaintext/ciphertext pair turns out to be useless. Considering the probability of this event we need \(2^{10}\) times more pairs, thus we eventually need \(2^{84.11}\) ciphertext pairs.

Since one structure enables to produce \(2^{54.56}\) plaintext pairs of the desired difference, we need to generate \(2^{29.55}\) structures (by using \(2^{29.55}\) distinct constants) and run the above key elimination procedure for all structures.

**Details of Key Elimination.** In the key elimination we combine several techniques to reduce the complexity. In particular we use the Difference Table. Its entry is indexed by \((x, x', y) \in \{(0,1)^4\}^3\) and the entry is a set \(K = \{k : y = S(k \oplus x) \oplus S(k \oplus x')\}\). We also use the relationships between subkeys induced from the key schedule, or we can directly guess the key if input and output pairs of the corresponding \(F\) are fixed (not only their differentials).

**Complexity Estimation.** For each plaintext-ciphertext pair, the procedure regarding \(K_2\) and \(K_3\) requires the 17 evaluations of \(F\) function, shown on the dotted lines in Figs 2 and 3, and \(F^{2^3,4,5,6}\), total 22 functions. This amounts to \(22/(23 \cdot 8)\) encryptions of 22-round TWINE. Consequently, we can attack 23-round TWINE-80 with the time complexity \(2^{50.11+10} \cdot 2^{20} \cdot 22/(23 \cdot 8) = 2^{77.04}\) encryptions, and the memory complexity \(2^{80}/64 = 2^{74}\) blocks.

![Fig. 2. The First 4 Rounds in the Impossible Differential Attack.](image)

### 4.3 Saturation Attack

Saturation attack [16] is also a powerful attack against GFS-based ciphers. We consider 4-bit-wise saturations. The state consists of \(2^4\) variables, denoted by \(S = (S_0, \ldots, S_{15})\), where \(S_i\) has the following four status (here \(X\) is the plaintext):

- **Constant** (\(C\)) : \(\forall i, j \quad X_i = X_j\)
- **All** (\(A\)) : \(\forall i, j \quad i \neq j \Rightarrow X_i \neq X_j\)
- **Balance** (\(B\)) : \(\bigoplus_{i=0}^{2^4-1} X_i = 0\)
- **Unknown** (\(U\)) : Others
Let $\alpha = (\alpha_0, \ldots, \alpha_{15})$ and $\beta = (\beta_0, \ldots, \beta_{15})$, $\alpha_i, \beta_i \in \{C, A, B, U\}$, be the initial state and the $t$-round state holding with probability 1. If we have $\alpha_i = A$ and $\beta_j \neq U$ for some $i$ and $j$, we call $\alpha \xrightarrow{tr} \beta$ an $t$-round saturation characteristic (SC).

TWINE has 15-round SC with input consisting of one $C$ and fifteen $A$s and output consisting of 4 $B$s (the remaining consists of $U$s), for example:

$$
\begin{align*}
(\overline{A}_{12}, C, \overline{A}_3) & \xrightarrow{15r} (\overline{U}_3, B, \overline{U}_5, B, U, B), \\
(\overline{A}_6, C, \overline{A}_9) & \xrightarrow{15r} (U, B, \overline{U}_3, B, U, B, \overline{U}_3, B, \overline{U}_4).
\end{align*}
$$

(6) (7)

Suppose we use SC of Eq. (7) to break 22-round TWINE-80. Let $S$-structure denote a set of $2^{60}$ plaintexts induced from Eq. (7), i.e., the input block $X_6$ is fixed to a constant and the other blocks take all combinations.

Our attack recovers a 72-bit subkey vector $K_{\text{target}} = (RK_{22}, RK_{21}[0,2,3,4,5,6,7], RK_{20}[0,6,7], RK_{16})$ based on the fact that (the state of) $X_1^{15}$ is $B$ with these 15-round SCs. Here, the state of $X_1^{15}$ being $B$ implies the coincidence of states between $X_1^{16}$ and $F_0^{16}$, the output of the leftmost $F$ function in the round 16, computed from the $S$-structure. From this, we calculate the sums of $F_0^{16}$ and $X_1^{16}$ independently, and choose a key that makes these values the same as a candidate for the correct key. The basic procedure for one $S$-structure is as follows.

1. Encrypt an $S$-structure and obtain $2^{60}$ ciphertexts. As our target is the 22-round version, the ciphertext is written as $X^{22}$.
2. List all ciphertexts except their rightmost (16-th) block, $X_{15}^{22}$. The result is denoted by List $L_1$. It is merely a set of 60-bit values and the values with even appearances need not be stored. We then guess

$$K_1 = (RK_{22}^{22}, RK_{[0,1,2,3,4,5,6]}, RK_{[2,3,4,5,7], 20}, RK_{17}, RK_{16})$$

Fig. 3. The Last 5 Rounds in the Impossible Differential Attack.
and compute the sum of outputs of $F_0^{16}$ (the leftmost $F$ function in the round 16) using all entries in $\mathcal{L}_1$ with each guess for $K_1$. We remark that a pair of an entry of $\mathcal{L}_1$ and a guess for $K_1$ uniquely determines the output of $F_0^{16}$. The key guesses are grouped according to the sum of $F_0^{16}$’s outputs.

Let $\mathcal{G}_1(s)$ be the key group with $F_0^{16}$ output sum $s \in \{0,1\}^4$.

3. Count the appearance of 48-bit ciphertext subsequences, $X_{[0,2,4,5,6,7,8,9,10,11,14,15]}^{22}$, and list those that have odd counts to form the list $\mathcal{L}_2$. We then guess

$$K_2 = (RK_2^{22}_{[2,3,4,5,7]}, RK_2^{21}_{[0,5,6]}, RK_7^{20}, RK_2^{18})$$

and compute the sum of $X_0^{16}$ using all entries in $\mathcal{L}_2$ with each guess for $K_2$. The key guesses are grouped according to the sum of $X_0^{16}$. The key group with $X_0^{16}$ sum being $s' \in \{0,1\}^4$ is denoted by $\mathcal{G}_2(s')$.

4. Extract the all “consistent” combinations from $\mathcal{G}_1(s)$ and $\mathcal{G}_2(s)$ for all $s \in \{0,1\}^4$, and output them as the set of valid key candidates for $K_{\text{target}}$. This can be done as follows. Let $v \in \mathcal{G}_1(0000)$ and $w \in \mathcal{G}_2(0000)$. We denote the guess for $RK_1^v$ in $v$ and $w$ by $RK_1^v (v)$ and $RK_1^v (w)$. Both $v$ and $w$ contain guesses of $RK_2^{22}_{[2,3,4,5]}$ and $RK_5^{21}$. If the following four equations,

\[
\begin{align*}
RK_2^{22}_{[2,3,4,5]}(v) &= RK_2^{22}_{[2,3,4,5]}(w), \\
RK_5^{21}(v) &= RK_5^{21}(w), \\
RK_7^{20}(w) &= S(RK_7^{20}(v)) \oplus RK_{21}^{48}(w), \\
RK_7^{21}(w) &= S(S(RK_7^{20}(w)) \oplus S^{-1}(RK_6^{20}(v) \oplus RK_{21}^{48}(v))) \oplus RK_5^{21}(w)
\end{align*}
\]

hold true, a valid key candidate for $K_{\text{target}}$ is obtained by combining $v$ and $w$. The check is done for all pairs from $\mathcal{G}_1(s) \times \mathcal{G}_2(s)$, and for all $s \in \{0,1\}^4$.

Taking Step 2 for example, we explain the detailed procedure. We first guess $RK_2^{22}$ and compute $X_1^{21}(= F_0^{21}(X_0^{22}))$ using $\mathcal{L}_1$ with $2^{64}$ $F$ evaluations. Then we substitute $X_1^{21}$ (8 bits) written in $\mathcal{L}_1$ with $X_0^{21}$ (4 bits) and obtain a list of 56-bit sequences, and collect the values with odd appearance to form a new list, called $\mathcal{L}_{1,1}$. Next, we guess $RK_2^{22}$ and compute $X_2^{21}$ based on the guess with $2^{64}$ $F$ evaluations. We then substitute $X_2^{21}$ (12 bits) written in $\mathcal{L}_{1,1}$ with $X_0^{21}$ and obtain the list of 52-bit sequences and collect the values with odd appearance to form a new list, called $\mathcal{L}_{1,2}$. The above procedure is repeated to gradually reduce the list size. Eventually the computation of the sum of $F_0^{16}$ outputs requires $2^{13.80}$ $F$ evaluations (equivalently $2^{26.34}$ encryptions of 22-round TWINE). A similar complexity reduction can also be applied to Step 3, however, the computation of Step 3 is much smaller than that of Step 2 (due to the small space for the key guess) in any case.

As checks are done w.r.t. 4-bit internal values, the above procedure with one $S$-structure rejects a wrong key candidate for $K_{\text{target}}$ with probability $1 - 2^{-4}$ (i.e. the size of candidates is reduced to 1/16), hence we basically need at least

\[3 \text{ More precisely, } RK_{[0,5]}^{20}, RK_{[1,7]}^{19}, RK_3^{18} \text{ are required to compute } F_0^{16} \text{ outputs. Also } RK_1^{20} \text{ and } RK_3^{19} \text{ are required to compute } X_0^{16} \text{ in Step 3. These RKs can be computed from } K_1 \text{ or } K_2. \text{ See Appendix E.} \]
18 $S$-structures to identify the right key. However, this is impossible as each sub-block is 4-bit. To elude the problem, we exploit the key schedule: the structure of the key schedule allows us to derive the 80-bit key with $2^{68}$ candidates for 72-bit subkey, and an exhaustive search for the remaining 8-bit subkey, and the final key check, which is trivial.

Summarizing, the attack with an $S$-structure requires $2^{60}$ plaintexts to be encrypted, and $2^{77}$ (which follows from $2^{66.34} + 2^{76} + \rho$, where $\rho$ denotes the computation of Step 3, which is negligible) encryptions. The memory complexity is $2^{67}$ (64-bit) blocks. If we want to further reduce the complexity, using multiple $S$-structures (using distinct constants with the same SC) can help. The result is shown by Table 4. According to our investigation, the attack with 5 or more structures has higher time complexity than that with 4 structures, hence the best one is with 4 structures.

Table 4. Complexity of Saturation Attack.

<table>
<thead>
<tr>
<th># of Struct.s</th>
<th>Data</th>
<th>Time (Enc)</th>
<th>Memory (Block)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$2^{60}$</td>
<td>$2^{77}$</td>
<td>$2^{67}$</td>
</tr>
<tr>
<td>2</td>
<td>$2^{61}$</td>
<td>$2^{73}$</td>
<td>$2^{67}$</td>
</tr>
<tr>
<td>3</td>
<td>$2^{61.59}$</td>
<td>$2^{68.97}$</td>
<td>$2^{67}$</td>
</tr>
<tr>
<td>4</td>
<td>$2^{62}$</td>
<td>$2^{68.43}$</td>
<td>$2^{67}$</td>
</tr>
</tbody>
</table>

4.4 Differential / Linear Cryptanalysis

To evaluate the resistance against differential cryptanalysis (DC) [5] and linear cryptanalysis (LC) [29], we need to know the number of differentially and linearly active S-boxes, denoted by $AS_D$ and $AS_L$, respectively. We performed a computer-based search for differential and linear paths, and evaluated $AS_D$ and $AS_L$ for each round. As a result, the numbers of $AS_D$ and $AS_L$ are the same, as shown by Table 5. Since our S-box has $2^{-2}$ maximum differential and linear probabilities, the maximum differential and linear characteristic probabilities are both $2^{-64}$ for 15 rounds. Examples of 14-round differential ($\Delta$) and linear ($\Gamma$) characteristics having the minimum I/O weights are as follows. They involve 30 active S-boxes, and thus the characteristic probability is $2^{-60}$.

$$\begin{align*}
\Delta &= (0^9, 1, 0, 1, 0, 1, 0, 0, 1, 1)^{14r} (0^3, 1, 0^4, 1, 0, 0, 1, 0, 0, 1, 1), \\
\Gamma &= (0^6, 1, 1, 0^3, 1, 0, 0, 1, 1)^{14r} (0^9, 1, 0^3, 1, 0, 1, 1). \tag{11}
\end{align*}$$

Here, 1 denotes an arbitrary non-zero difference (mask) and 0 denotes the zero difference (mask) for $\Delta$ ($\Gamma$). Compared to the impossible differential attack, we naturally expect the key recovery attacks exploiting the key schedule with these
differential/linear characteristics are less powerful, since 14-round impossible differential characteristic has much fewer (only 2) weights, and fewer weights imply the more attackable rounds in the key guessing. We remark that a computer-based search for the maximum differential probability (rather than the characteristic probability) of GFS was proposed by [30]. However, applying their algorithm to our 16-block case seems infeasible due to the computational complexity.

<table>
<thead>
<tr>
<th>Round</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>$AS_D, AS_L$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>11</td>
<td>14</td>
<td>18</td>
<td>22</td>
<td>24</td>
<td>27</td>
<td>30</td>
<td>32</td>
</tr>
</tbody>
</table>

4.5 Key Schedule-based Attacks

Related-Key Differential Attacks. The related-key attack, proposed by Biham [4], is an attack applicable to the environment where the adversary can somehow modify the key input. We focus on the typical setting, that is, the adversary is allowed to insert a key differential. In order to evaluate the resistance of TWINE against the related-key attack, we implemented the search method proposed by Biryukov et al. [8], which counts the number of active S-boxes for combined data processing and key schedule parts. See [8] for the detail of the algorithm. We (naturally) searched 4-bit truncated differential paths. As S-box has maximum differential probability being $2^{-2}$, we needed 40 (64) active S-boxes for TWINE-80 (TWINE-128).

Due to the computational constraint the full-search is only feasible for TWINE-80. As a result, we confirmed that the number of active S-boxes reaches 40 for the 22-round. Appendix F provides the number of active S-boxes and the corresponding truncated differential paths.

Other Attacks. For the slide attack [7], the key schedule of TWINE inserts distinct constants for each round. This is a typical way to thwart slide attacks and hence we consider TWINE is immune to the slide attack.

For Meet-In-The-Middle (MITM) attack, we confirmed that the round keys for the first 3 (5) rounds contain all key bits for the 80-bit (128-bit) key case. Thus, we consider it is difficult to mount the basic MITM attack against the full-round TWINE. Note that the recently-proposed MITM variant, called biclique attack [11], may work even when all key bits are used in the relatively small number of rounds. The evaluation of such attack against TWINE is a future topic.

The result of our security evaluation for TWINE is summarized at Table 6.
Table 6. Summary of Attacks on TWINE.

<table>
<thead>
<tr>
<th>Key (bits)</th>
<th>Attack</th>
<th>Rounds</th>
<th>Data (blocks)</th>
<th>Time (encryption)</th>
<th>Memory (blocks)</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>Impossible Diff.</td>
<td>23</td>
<td>$2^{61.39}$</td>
<td>$2^{68.38}$</td>
<td>$2^{74}$</td>
</tr>
<tr>
<td></td>
<td>Saturation</td>
<td>22</td>
<td>$2^{62}$</td>
<td>$2^{68.43}$</td>
<td>$2^{77}$</td>
</tr>
<tr>
<td>128</td>
<td>Impossible Diff.</td>
<td>24</td>
<td>$2^{52.21}$</td>
<td>$2^{115.10}$</td>
<td>$2^{118}$</td>
</tr>
<tr>
<td></td>
<td>Saturation</td>
<td>23</td>
<td>$2^{62.81}$</td>
<td>$2^{106.14}$</td>
<td>$2^{103}$</td>
</tr>
</tbody>
</table>

5 Implementation

5.1 Hardware

We implemented TWINE on ASIC using a 90nm standard cell library with logic synthesis done by Synopsys Design Compiler Version D-2010.03-SP1-1. Following the recent trend in the lightweight implementations [9,13], the figures are shown for the case when Scan Flip-Flops (FFs) are used. In our library, a D-FF and 2-to-1 MUX cost 5.5 GE and 2.25 GE, and a Scan FF costs 6.75 GE; hence this technique saves 1.0 GE per 1-bit storage.

The data path of TWINE-80 encryption circuit is in Fig. 4, and the implementation result is shown by Table 7. We also show the detail of TWINE-80 encryption implementation in Table 8. The figures must be taken with cares, because they are related to the type of memory unit (FF), technology, library, etc, as pointed out by [13]. Even single XOR gate has several grades, from fast-but-large and slow-but-small. As suggested by [13], we list Gates/Memory Bit in the table, which denotes the size (in Gate Equivalent (GE)) of 1-bit memory device used for the key and states.

We did not perform a thorough logic minimization of the S-box circuit, which currently costs 30 GEs. The S-box logic minimization can further reduce the size.

We also tried a serialized implementation. Though it is not yet finished, the preliminary result indicates that encryption-only TWINE-80 can be implemented 1,116 GEs. The details of our serialized implementation will be given in a forthcoming paper.

5.2 Software

To evaluate the performance on embedded software, we implement TWINE on Atmel AVR 8-bit Micro-controller. The target device is ATmega163, which has 16K bytes Flash, 512 bytes EEPROM and 1024 bytes SRAM. We implemented the four versions: speed-first, ROM-first (minimizing the consumption), and RAM-first, and the double-block implementation, where two message blocks are processed in parallel. Such an implementation works for parallelizable modes of operations, e.g., the counter mode and PMAC. All implementations include the precomputation of round keys, i.e. they do not use an on-the-fly key schedule.
Table 7. ASIC Implementation Results. For some implementations, the figures of Throughput and Cycles/Block is an estimated value.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Function</th>
<th>Block size (bit)</th>
<th>Key size (bit)</th>
<th>Cycles/Block (Kbps@100KHz)</th>
<th>Throughput (GE)</th>
<th>Area (GE)</th>
<th>Gates / Memory bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>TWINE</td>
<td>Enc</td>
<td>64</td>
<td>80</td>
<td>36</td>
<td>178</td>
<td>1503</td>
<td>6.75 round</td>
</tr>
<tr>
<td>TWINE</td>
<td>Enc+Dec</td>
<td>64</td>
<td>80</td>
<td>36</td>
<td>178</td>
<td>1799</td>
<td>6.75 round</td>
</tr>
<tr>
<td>TWINE</td>
<td>Enc</td>
<td>64</td>
<td>128</td>
<td>36</td>
<td>178</td>
<td>1866</td>
<td>6.75 round</td>
</tr>
<tr>
<td>TWINE</td>
<td>Enc+Dec</td>
<td>64</td>
<td>128</td>
<td>36</td>
<td>178</td>
<td>2285</td>
<td>6.75 round</td>
</tr>
<tr>
<td>PRESENT [36]</td>
<td>Enc</td>
<td>64</td>
<td>80</td>
<td>540</td>
<td>11.8</td>
<td>1116</td>
<td>6.75 serial</td>
</tr>
<tr>
<td>PRESENT [9]</td>
<td>Enc</td>
<td>64</td>
<td>80</td>
<td>32</td>
<td>547</td>
<td>1000</td>
<td>n/a serial</td>
</tr>
<tr>
<td>TWINE</td>
<td>Enc</td>
<td>64</td>
<td>80</td>
<td>540</td>
<td>11.8</td>
<td>1116</td>
<td>6.75 serial</td>
</tr>
<tr>
<td>PRESENT [36]</td>
<td>Enc</td>
<td>64</td>
<td>80</td>
<td>547</td>
<td>11.4</td>
<td>1090</td>
<td>n/a serial</td>
</tr>
<tr>
<td>PRESENT [9]</td>
<td>Enc</td>
<td>64</td>
<td>80</td>
<td>32</td>
<td>200</td>
<td>3200</td>
<td>6 round</td>
</tr>
<tr>
<td>AES [31]</td>
<td>Enc</td>
<td>128</td>
<td>128</td>
<td>226</td>
<td>57</td>
<td>2400</td>
<td>6 serial</td>
</tr>
<tr>
<td>mCRYPTON [27]</td>
<td>Enc</td>
<td>64</td>
<td>64</td>
<td>13</td>
<td>492.3</td>
<td>2420</td>
<td>6 round</td>
</tr>
<tr>
<td>HIGHT [21]</td>
<td>Enc+Dec</td>
<td>64</td>
<td>128</td>
<td>34</td>
<td>188.25</td>
<td>3048</td>
<td>n/a round</td>
</tr>
<tr>
<td>KLEIN [19]</td>
<td>Enc</td>
<td>64</td>
<td>80</td>
<td>17</td>
<td>376.4</td>
<td>2629</td>
<td>n/a round</td>
</tr>
<tr>
<td>KLEIN [19]</td>
<td>Enc+Dec</td>
<td>64</td>
<td>80</td>
<td>21</td>
<td>23.6</td>
<td>1178</td>
<td>n/a serial</td>
</tr>
<tr>
<td>DESL [26]</td>
<td>Enc</td>
<td>64</td>
<td>56</td>
<td>144</td>
<td>44.4</td>
<td>2309</td>
<td>12.19 serial</td>
</tr>
<tr>
<td>DESL [26]</td>
<td>Enc</td>
<td>64</td>
<td>56</td>
<td>144</td>
<td>44.4</td>
<td>1848</td>
<td>12.19 serial</td>
</tr>
<tr>
<td>KATAN [13]</td>
<td>Enc</td>
<td>64</td>
<td>80</td>
<td>254</td>
<td>25.1</td>
<td>1054</td>
<td>6.25 serial</td>
</tr>
<tr>
<td>Piccolo [40]</td>
<td>Enc</td>
<td>64</td>
<td>80</td>
<td>27</td>
<td>2.37</td>
<td>1496</td>
<td>6.25 round</td>
</tr>
<tr>
<td>Piccolo [40]</td>
<td>Enc+Dec</td>
<td>64</td>
<td>80</td>
<td>27</td>
<td>2.37</td>
<td>1634</td>
<td>6.25 round</td>
</tr>
<tr>
<td>Piccolo [40]</td>
<td>Enc</td>
<td>64</td>
<td>80</td>
<td>432</td>
<td>14.8</td>
<td>1043</td>
<td>6.25 serial</td>
</tr>
<tr>
<td>Piccolo [40]</td>
<td>Enc+Dec</td>
<td>64</td>
<td>80</td>
<td>432</td>
<td>14.8</td>
<td>1103</td>
<td>6.25 serial</td>
</tr>
<tr>
<td>LED [18]</td>
<td>Enc</td>
<td>64</td>
<td>80</td>
<td>1872</td>
<td>3.4</td>
<td>1040</td>
<td>6/4.67s serial</td>
</tr>
<tr>
<td>PRINTcipher [25]</td>
<td>Enc</td>
<td>48</td>
<td>80</td>
<td>48</td>
<td>12.5</td>
<td>503*</td>
<td>n/a round</td>
</tr>
</tbody>
</table>

† Gate Equivalent : cell area/2-input NAND gate size (2.82).
¶ Includes a key register that costs 360 GEs; Piccolo can be implemented without a key register if key signal holds while encryption.
⋄ Mixed usage of two memory units.
⋆ Hardwired key.

Table 8. TWINE-80 Encryption Hardware Implementation.

<table>
<thead>
<tr>
<th>Data Processing Part (GE)</th>
<th>Key Scheduling Part (GE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Register</td>
<td>432</td>
</tr>
<tr>
<td>S-box</td>
<td>240</td>
</tr>
<tr>
<td>Round Key XOR</td>
<td>64</td>
</tr>
<tr>
<td>S-box out XOR</td>
<td>64</td>
</tr>
<tr>
<td>RC register</td>
<td>33</td>
</tr>
<tr>
<td>Others/Control</td>
<td>34</td>
</tr>
</tbody>
</table>

16
In the speed-first implementation, two rounds are processed in one loop. This removes the block shuffle between the first and second rounds. A further speeding up is possible if more rounds are contained in one loop at the cost of increased memory. Our program keeps all 4-bit blocks in the distinct registers. RAM load instruction (LD) is faster than ROM load instruction (LPM), hence the S-box and the constants are stored at RAM. The data arrangement is carefully considered to avoid carry in the address computation. The double-block implementation stores the S-boxes in ROM.

Our result is in Table 9, and a comparison is in Table 10. In Table 10 we list the (scaled) throughput/code ratio for a performance measure (See Table 10 for the formula), following [35]. AES’s performance is still quite impressive, however, one can also observe a good performance of TWINE.

One might be interested in the performance of TWINE under 32/64-bit CPU. We are currently working on this, in particular using the vector permutation instructions, which was shown to be very powerful for AES [20].

Table 9. Software Implementation of TWINE on ATmega163.

<table>
<thead>
<tr>
<th>Target</th>
<th>Key schedule (cycles)</th>
<th>Encryption (cycles/block)</th>
<th>Decryption (cycles/block)</th>
<th>ROM (bytes)</th>
<th>RAM (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed-first</td>
<td>2,170</td>
<td>2,165</td>
<td>2,166</td>
<td>1,304</td>
<td>414</td>
</tr>
<tr>
<td>ROM-first</td>
<td>12,022</td>
<td>18,794</td>
<td>18,689</td>
<td>728</td>
<td>335</td>
</tr>
<tr>
<td>RAM-first</td>
<td>12,058</td>
<td>18,794</td>
<td>18,688</td>
<td>792</td>
<td>191</td>
</tr>
<tr>
<td>Double-block</td>
<td>1,887</td>
<td>1,301</td>
<td>1,302</td>
<td>2,294</td>
<td>386</td>
</tr>
</tbody>
</table>
### Table 10. Comparison of Software Implementation on AVR.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Key (bits)</th>
<th>Block (bits)</th>
<th>Language</th>
<th>ROM (bytes)</th>
<th>RAM (bytes)</th>
<th>Enc (cyc/byte)</th>
<th>Dec (cyc/byte)</th>
<th>ETput (bits)</th>
<th>DTput (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TWINE</td>
<td>80</td>
<td>64</td>
<td>asm</td>
<td>1,304</td>
<td>414</td>
<td>271</td>
<td>271</td>
<td>2.14</td>
<td>2.14</td>
</tr>
<tr>
<td>PRESENT [33]</td>
<td>80</td>
<td>64</td>
<td>asm</td>
<td>2,398</td>
<td>528</td>
<td>1,199</td>
<td>1,228</td>
<td>0.28</td>
<td>0.28</td>
</tr>
<tr>
<td>PRESENT [17]</td>
<td>80</td>
<td>64</td>
<td>N/A</td>
<td>936</td>
<td>0</td>
<td>1,340</td>
<td>1,405</td>
<td>0.80</td>
<td>0.76</td>
</tr>
<tr>
<td>DES [17]</td>
<td>56</td>
<td>64</td>
<td>N/A</td>
<td>4,314</td>
<td>0</td>
<td>1,079</td>
<td>1,019</td>
<td>0.21</td>
<td>0.22</td>
</tr>
<tr>
<td>DESXL [17]</td>
<td>184</td>
<td>64</td>
<td>N/A</td>
<td>3,192</td>
<td>0</td>
<td>1,066</td>
<td>995</td>
<td>0.31</td>
<td>0.31</td>
</tr>
<tr>
<td>HIGHT [17]</td>
<td>128</td>
<td>64</td>
<td>N/A</td>
<td>5,672</td>
<td>0</td>
<td>371</td>
<td>371</td>
<td>0.48</td>
<td>0.48</td>
</tr>
<tr>
<td>IDEA [17]</td>
<td>128</td>
<td>64</td>
<td>N/A</td>
<td>596</td>
<td>0</td>
<td>338</td>
<td>1,924</td>
<td>4.97</td>
<td>4.97</td>
</tr>
<tr>
<td>TEA [17]</td>
<td>128</td>
<td>64</td>
<td>N/A</td>
<td>1,140</td>
<td>0</td>
<td>784</td>
<td>784</td>
<td>1.11</td>
<td>1.11</td>
</tr>
<tr>
<td>SEA [17]</td>
<td>96</td>
<td>96</td>
<td>N/A</td>
<td>2,132</td>
<td>0</td>
<td>805</td>
<td>805</td>
<td>0.58</td>
<td>0.58</td>
</tr>
<tr>
<td>AES [12]</td>
<td>128</td>
<td>128</td>
<td>asm</td>
<td>1,912</td>
<td>432</td>
<td>125</td>
<td>181</td>
<td>3.42</td>
<td>2.35</td>
</tr>
</tbody>
</table>

1 Encryption Throughput per Code: \((1/\text{Enc})/(\text{ROM} + \text{RAM})\) (scaled by 10^6)
2 Decryption Throughput per Code: \((1/\text{Dec})/(\text{ROM} + \text{RAM})\) (scaled by 10^6)

### 6 Conclusions

We have presented a lightweight block cipher TWINE, which has 64-bit block and 80 or 128-bit key. It is primarily designed to fit extremely-small hardware, yet provides a notable performance under embedded software. This characteristic mainly originates from the Type-2 generalized Feistel with a highly-diffusive block shuffle. We performed a thorough security analysis, in particular for the impossible differential and saturation attacks. Although the result implies the sufficient security of full-round TWINE, its security naturally needs to be studied further.

**Acknowledgments.** The authors would like to thank the anonymous reviewers for many useful comments. We thank Maki Shigeri, Etsuko Tsujihara, and Teruo Saito for discussions, and Daisuke Ikemura for investigation on hardware-related issues.

### References

   /Software_Implementation_of_Block_Cipher_PRESENT_for_8-Bit_Platforms


A  Key Schedule for 128-bit Key

Algorithm A.1: TWINE.KeySchedule-128($K_{(128)}$, $RK_{(32\times36)}$)

\[
\begin{align*}
WK_{(128)} & \leftarrow K \\
WK_{0(16)}||WK_{1(16)}|| \ldots ||WK_{6(16)}||WK_{7(16)} & \leftarrow WK \\
RK_{0(4)} & \leftarrow WK_{2}, \; RK_{1(4)} \leftarrow WK_{3}, \; RK_{2(4)} \leftarrow WK_{12}, \; RK_{3(4)} \leftarrow WK_{15} \\
RK_{4(4)} & \leftarrow WK_{17}, \; RK_{5(4)} \leftarrow WK_{18}, \; RK_{6(4)} \leftarrow WK_{28}, \; RK_{7(4)} \leftarrow WK_{31} \\
RK_{32(32)} & \leftarrow RK_{0}||RK_{1}|| \ldots ||RK_{6}||RK_{7}
\end{align*}
\]

for $i$ ← 2 to 36

\[
\begin{align*}
WK_{i} & \leftarrow WK_{i} \oplus S(WK_{0}) \\
WK_{4} & \leftarrow WK_{4} \oplus S(WK_{16}) \\
WK_{23} & \leftarrow WK_{23} \oplus S(WK_{30}) \\
WK_{7} & \leftarrow WK_{7} \oplus 0||CON_{H}^{-1} \\
WK_{19} & \leftarrow WK_{19} \oplus 0||CON_{L}^{-1}
\end{align*}
\]

tmp_{0} ← WK_{0}, \; tmp_{1} ← WK_{1}, \; tmp_{2} ← WK_{2}, \; tmp_{3} ← WK_{3}

do

\[
\begin{align*}
WK_{j+4} & \leftarrow WK_{j+4} \oplus WK_{j+6} \\
WK_{j+4+1} & \leftarrow WK_{j+4+3} \oplus WK_{j+4+7} \\
WK_{28} & \leftarrow tmp_{1}, \; WK_{29} \leftarrow tmp_{2}, \; WK_{30} \leftarrow tmp_{3}, \; WK_{31} \leftarrow tmp_{3} \\
RK_{0} & \leftarrow WK_{2}, \; RK_{1} \leftarrow WK_{3} \\
RK_{6} & \leftarrow WK_{12}, \; RK_{7} \leftarrow WK_{15} \\
RK_{1(32)} & \leftarrow RK_{0(4)}||RK_{1(4)}|| \ldots ||RK_{6(4)}||RK_{7(4)}
\end{align*}
\]

B  Test Vectors

Table 11. Test Vectors in the Hexadecimal Notation.

<table>
<thead>
<tr>
<th>key length</th>
<th>80-bit</th>
<th>128-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>00112233 44556677 8899</td>
<td>00112233 44556677 8899AABB CCDDDEEFF</td>
</tr>
<tr>
<td>plaintext</td>
<td>01234567 89ABCDEF</td>
<td>01234567 89ABCDEF</td>
</tr>
<tr>
<td>ciphertext</td>
<td>7C1F0F80 B1DF9C28</td>
<td>979FF9B3 79B5A988</td>
</tr>
</tbody>
</table>

C  80-bit Key Schedule

Figure 5 depicts the key schedule for TWINE-80.
D Supplementary Information for Impossible Differential Cryptanalysis

The following is the subkey relationships used for the impossible differential cryptanalysis.

\[ \begin{align*}
RK_3 &= RK_1 \\
RK_5 &= RK_1 \\
RK_4 &= RK_6 \\
RK_0^{21} &= S[S[RK_4^{19}] \oplus RK_4^1] \oplus RK_4^{21}] \oplus RK_7 \\
RK_0^{21} &= S^{-1}[S^{-1}[RK_0^{21}] \oplus S[RK_0^{20}] \oplus RK_1] \oplus RK_3] \\
RK_0^{22} &= RK_4^{19} \\
RK_2^2 &= S[RK_2^0] \oplus S[RK_2^1] \oplus RK_2^2 \\
RK_2^{22} &= S[S[S[RK_4^0] \oplus RK_4^1] \oplus RK_2^{22}] \oplus S[S[S[RK_4^0] \oplus S[RK_2^2] \oplus RK_2^1] \oplus S^{-1}[RK_2^{22}] \\
&\quad \oplus S^{-1}[S^{-1}[RK_2^{22}] \oplus S[RK_2^{20}] \oplus RK_1] \oplus RK_3] \oplus S[RK_2^{20}] \oplus RK_2^1] \oplus S[RK_2^{20}] \oplus RK_2^1] \oplus RK_2^1] \oplus RK_2^1] \\
RK_6^{22} &= S[S[RK_6^0] \oplus S[RK_6^1] \oplus S^{-1}[RK_6^{22}] \oplus S^{-1}[S^{-1}[RK_6^{22}] \oplus S[RK_6^0] \oplus RK_1] \\
&\quad \oplus RK_1] \oplus S[RK_6^{21}] \oplus RK_3] \\
RK_0^{23} &= S[S[RK_4^{21}] \oplus RK_4^2] \oplus RK_2^{23}] \oplus S^{-1}[S[RK_4^0] \oplus S[RK_2^2] \oplus RK_1] \oplus RK_3] \oplus S[RK_2^{20}] \oplus RK_2^1] \oplus S[RK_2^{20}] \oplus RK_2^1] \oplus RK_2^1] \oplus RK_2^1] \\
RK_2^{23} &= S^{-1}[S^{-1}[RK_2^{23}] \oplus S[RK_2^{20}] \oplus RK_1] \oplus RK_3] \\
RK_4^{23} &= RK_3^{20} \\
RK_5^{23} &= RK_2^{21} \\
RK_6^{23} &= S[RK_3^{21}] \oplus S[RK_3^2] \oplus S^{-1}[RK_2^2 \oplus RK_3] \\
RK_7^{23} &= S[S[RK_2^0] \oplus S[RK_2^{22}] \oplus S[RK_3^2] \oplus RK_2^1] \oplus RK_3^1] \oplus RK_4^{19}
\end{align*} \]

Fig. 5. 80-bit Key Schedule.
Table 12. Procedure of Round Key Determination.

<table>
<thead>
<tr>
<th>Step</th>
<th>Target Key</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>RK₀, RK₁, RK₂</td>
<td>diff-table</td>
<td>Input of F₁ and output of F₂ are decided from diff-table.</td>
</tr>
<tr>
<td>2</td>
<td>RK₃, RK₄</td>
<td>key rel.</td>
<td>Input of F₁ is decided from diff-table.</td>
</tr>
<tr>
<td>3</td>
<td>RK₅</td>
<td>guess</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>RK₆</td>
<td>IO value</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>RK₇</td>
<td>guess</td>
<td>Input of F₂ is decided.</td>
</tr>
<tr>
<td>6</td>
<td>RK₈</td>
<td>diff-table</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>RK₉</td>
<td>guess</td>
<td>Input of F₂ is decided.</td>
</tr>
<tr>
<td>8</td>
<td>RK₁₀</td>
<td>key rel.</td>
<td>Output of F₀ is decided.</td>
</tr>
<tr>
<td>9</td>
<td>RK₁₁</td>
<td>diff-table</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>RK₁₂</td>
<td>guess</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>RK₁₃</td>
<td>IO value</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>RK₁₄, RK₁₅, RK₁₆</td>
<td>diff-table</td>
<td>IO value of F₀ is decided.</td>
</tr>
<tr>
<td>13</td>
<td>RK₁₇, RK₁₈</td>
<td>key rel.</td>
<td>Input of F₀ is decided from diff-table.</td>
</tr>
<tr>
<td>14</td>
<td>RK₂₀, RK₂₁, RK₂₂</td>
<td>key rel.</td>
<td>Input of F₀ is decided, then output of F₀ is decided.</td>
</tr>
<tr>
<td>15</td>
<td>RK₂₃</td>
<td>diff-table</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>RK₂₄</td>
<td>key rel.</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>RK₂₅</td>
<td>key rel.</td>
<td>IO value of F₂ is decided.</td>
</tr>
<tr>
<td>18</td>
<td>RK₂₆</td>
<td>diff-table</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>RK₂₇</td>
<td>IO value</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>RK₂₈</td>
<td>key rel.</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>RK₂₉</td>
<td>IO value</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>RK₃₀</td>
<td>guess</td>
<td>IO value of F₂ is decided.</td>
</tr>
<tr>
<td>23</td>
<td>RK₃₁</td>
<td>diff-table</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>RK₃₂, RK₃₃</td>
<td>key rel.</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>RK₃₄, RK₃₅, RK₃₆</td>
<td>key rel.</td>
<td></td>
</tr>
</tbody>
</table>

23
E Subkey Relationships used for Saturation Attack

The following is the subkey relationships used for the saturation attack.

\[
\begin{align*}
RK_3^{21} &= RK_3^{18} \\
RK_5^{21} &= RK_1^{19} \\
RK_2^{21} &= S[RK_2^{21}] \oplus RK_2^{18} \\
RK_2^{22} &= RK_7^{19} \\
RK_5^{22} &= RK_5^{20} \\
RK_3^{22} &= RK_3^{19} \\
RK_3^{22} &= RK_1^{20} \\
RK_6^{22} &= S^{-1}[RK_2^{21} \oplus RK_5^{20}] \\
RK_7^{22} &= S[S[RK_7^{20}] \oplus S^{-1}[RK_6^{20} \oplus RK_7^{17}]] \oplus RK_6^{21}
\end{align*}
\]

F Related-key Truncated Differential and Its Active S-box Numbers

Table 13 shows the number of active S-boxes using related-key differential, where \(\Delta KS\), \(\Delta RK\), \(\Delta X\) and \(AS\) denote key state difference, subkey difference, data difference, and the number of active S-boxes.

<table>
<thead>
<tr>
<th>Round</th>
<th>(\Delta KS)</th>
<th>(\Delta RK)</th>
<th>(\Delta X)</th>
<th>AS</th>
<th>Round</th>
<th>(\Delta KS)</th>
<th>(\Delta RK)</th>
<th>(\Delta X)</th>
<th>AS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4D010</td>
<td>A2</td>
<td>A256</td>
<td>0</td>
<td>12</td>
<td>60402</td>
<td>80</td>
<td>A0E2</td>
<td>22</td>
</tr>
<tr>
<td>2</td>
<td>D8108</td>
<td>E1</td>
<td>6931</td>
<td>6</td>
<td>13</td>
<td>0402C</td>
<td>05</td>
<td>A630</td>
<td>27</td>
</tr>
<tr>
<td>3</td>
<td>010C3</td>
<td>08</td>
<td>9896</td>
<td>8</td>
<td>14</td>
<td>C02C0</td>
<td>88</td>
<td>8D39</td>
<td>30</td>
</tr>
<tr>
<td>4</td>
<td>10C30</td>
<td>46</td>
<td>4462</td>
<td>9</td>
<td>15</td>
<td>02C01</td>
<td>10</td>
<td>5A2E</td>
<td>33</td>
</tr>
<tr>
<td>5</td>
<td>0C302</td>
<td>20</td>
<td>2288</td>
<td>10</td>
<td>16</td>
<td>2C010</td>
<td>22</td>
<td>62C3</td>
<td>35</td>
</tr>
<tr>
<td>6</td>
<td>C3020</td>
<td>94</td>
<td>9411</td>
<td>11</td>
<td>17</td>
<td>C0104</td>
<td>80</td>
<td>8191</td>
<td>38</td>
</tr>
<tr>
<td>7</td>
<td>30201</td>
<td>40</td>
<td>0968</td>
<td>14</td>
<td>18</td>
<td>01041</td>
<td>08</td>
<td>0824</td>
<td>38</td>
</tr>
<tr>
<td>8</td>
<td>02016</td>
<td>12</td>
<td>1306</td>
<td>15</td>
<td>19</td>
<td>10410</td>
<td>42</td>
<td>4202</td>
<td>39</td>
</tr>
<tr>
<td>9</td>
<td>20160</td>
<td>0C</td>
<td>4545</td>
<td>19</td>
<td>20</td>
<td>04102</td>
<td>00</td>
<td>0081</td>
<td>39</td>
</tr>
<tr>
<td>10</td>
<td>01604</td>
<td>00</td>
<td>108C</td>
<td>20</td>
<td>21</td>
<td>41020</td>
<td>84</td>
<td>8100</td>
<td>41</td>
</tr>
<tr>
<td>11</td>
<td>16040</td>
<td>58</td>
<td>0840</td>
<td>21</td>
<td>22</td>
<td>10208</td>
<td>41</td>
<td>4124</td>
<td>41</td>
</tr>
</tbody>
</table>