The SKINNY Family of Block Ciphers

Christof Beierle¹, Jérémy Jean², Stefan Kölbl³, Gregor Leander¹, Amir Moradi¹, Thomas Peyrin², Yu Sasaki⁴, Pascal Sasdrich¹, and Siang Meng Sim²

¹ Horst Görtz Institute for IT Security, Ruhr-Universität Bochum, Germany
² School of Physical and Mathematical Sciences, Nanyang Technological University, Singapore
³ DTU Compute, Technical University of Denmark, Denmark
⁴ NTT Secure Platform Laboratories, Japan

Abstract. We present a new tweakable block cipher family SKINNY, whose goal is to compete with NSA recent design SIMON in terms of hardware/software performances, while proving in addition much stronger security guarantees with regards to differential/linear attacks. In particular, unlike SIMON, we are able to provide strong bounds for all versions, and not only in the single-key model, but also in the related-key or related-tweak model. SKINNY has flexible block/key/tweak sizes and can also benefit from very efficient threshold implementations for side-channel protection. Regarding performances, it outperforms all known ciphers for ASIC round-based implementations, while still reaching an extremely small area for serial implementations and a very good efficiency for software and micro-controllers implementations (SKINNY has the smallest total number of AND/OR/XOR gates used for encryption process).

Key words: lightweight encryption, low-latency, tweakable block cipher, MILP.

1 Introduction

Due to the increasing importance of pervasive computing, lightweight cryptography is currently a very active research domain in the symmetric-key cryptography community. In particular, we have recently seen the apparition of many (some might say too many) lightweight block ciphers, hash functions and stream ciphers. While the term lightweight is not strictly defined, it most often refers to a primitive that allows compact implementations, i.e. minimizing the area required by the implementation. While the focus on area is certainly valid with many applications, most of them require additional performance criteria to be taken into account. In particular, the throughput of the primitive represents an important dimension for many applications. Besides that, power (in particular for passive RFID tags) and energy (for battery-driven device) may be major aspects.

Moreover, the efficiency on different hardware technologies (ASIC, FPGA) needs to be taken into account, and even micro-controllers become a scenario of importance. Finally, as remarked in [3], software implementations should not be completely ignored for these lightweight primitives, as in many applications the tiny devices will communicate with servers handling thousands or millions of them. Thus, even so research started by focusing on chip area only, lightweight cryptography is indeed an inherent multidimensional problem.

Investigating the recent proposals in more detail, a major distinction is eye-catching and one can roughly split the proposals in two classes. The first class of ciphers uses very strong, but less efficient

---

A more detailed version of this work appears in the proceedings of CRYPTO 2016 [2].
components (like the Sbox used in PRESENT [4] or LED [9], or the MDS diffusion matrix in LED or PICCOLO [19]). The second class of designs uses very efficient, but rather weak components (like the very small KATAN [5] or SIMON [1] round function).

From a security viewpoint, the analysis of the members of the first class can be conducted much easily and it is usually possible to derive strong arguments for their security. However, while the second class strategy usually gives very competitive performance figures, it is much harder with state-of-the-art analysis techniques to obtain security guarantees even with regards to basic linear or differential cryptanalysis. In particular, when using very light round functions, bounds on the probabilities of linear or differential characteristics are usually both hard to obtain and not very strong. As a considerable fraction of the lightweight primitives proposed got quickly broken within a few months or years from their publication date, being able to give convincing security arguments turns out to be of major importance.

Of special interest, in this context, is the recent publication of the SIMON and SPECK family of block ciphers by the NSA [1]. Those ciphers brought a huge leap in terms of performances. As of today, these two primitives have an important efficiency advantage against all its competitors, in almost all implementation scenarios and platforms. However, even though SIMON or SPECK are quite elegant and seemingly well-crafted designs, these efficiency improvements came at an essential price. Echoing the above, since the ciphers have a very light round function, their security bounds regarding classical linear or differential cryptanalysis are not so impressive, quite difficult to obtain or even non-existent. For example, in [13] the authors provide differential/linear bounds for SIMON, but, as we will see, one needs a big proportion of the total number of rounds to guarantee its security according to its block size. Even worse, no bound is currently known in the related-key model for any version of SIMON and thus there is a risk that good related-key differential characteristics might exist for this family of ciphers (while some lightweight proposals such as LED [9], PICCOLO [19] or some versions of TWINE [20] do provide such a security guarantee). One should be further cautious as these designs come from a governmental agency which does not provide specific details on how the primitives were built. No cryptanalysis was ever provided by the designers. Instead, the important analysis work was been carried out by the research community in the last few years and one should note that so far SIMON or SPECK remain unbroken.

It is therefore a major challenge for academic research to design a cipher that can compete with SIMON’s performances and additionally provides the essential strong security guarantees that SIMON is clearly lacking. We emphasize that this is both a research challenge and, in view of NSA’s efforts to propose SIMON into an ISO standard, a challenge that has likely a practical impact.

Lightweight Tweakable Block Ciphers and Side-Channel Protected Implementations. We note that tiny devices are more prone to be deployed into insecure environments and thus side-channel protected implementations of lightweight encryption primitives is a very important aspect that should be taken care of. One might even argue that instead of comparing performances of unprotected implementations of these lightweight primitives, one should instead compare protected variants (this is the recent trend followed by ciphers like ZORRO [8] or PICARO [17] and has actually already been taken into account long before by the cipher NOEKEON [7]). One extra protection against side-channel attacks can be the use of leakage resilient designs and notably through an extra tweak input of the cipher. Such tweakable block ciphers are rather rare, the only such candidate being Joltik-BC [11] or the internal cipher from Scream [21]. Coming up with a tweakable block cipher is indeed not an easy task as one must be extremely careful how to include this extra input that can be fully controlled by the attacker.

Our Contributions. We introduce a new lightweight family of block ciphers: SKINNY. Our goal here is to provide a competitor to SIMON in terms of hardware/software performances, while proving in addition much stronger security guarantees with regard to classical attacks.

With SKINNY, we have pushed further the recent trend of having a SPN cipher with locally non-optimal internal components: SKINNY is an SPN cipher that uses a compact Sbox, a new very sparse

---

1Actually, this separation is not only valid for lightweight designs. It can well be extended to more classical ciphers or hash functions as well.
diffusion layer, and a new very light key schedule. Yet, by carefully choosing our components and how they interact, our construction manages to retain very strong security guarantees. For all the SKINNY versions, we are able to prove using mixed integer linear programming (MILP) very strong bounds with respect to differential/linear attacks, not only in the single-key model, but also in the much more involved related-key model. Some versions of SKINNY have a very large key size compared to its block size and this theoretically renders the bounds search space huge. Therefore, the MILP methods we have devised to compute these bounds for a SKINNY-like construction can actually be considered a contribution by itself. As we will see later, compared to SIMON, in the single-key model SKINNY needs a much lower proportion of its total number of rounds to provide a sufficient bound on the best differential/linear characteristic. In the related-key model, the situation is even more at SKINNY’s advantage as no such bound is known for any version of SIMON as of today.

With regard to performance, SKINNY reaches very small area with serial ASIC implementations, yet it is actually the very first block cipher that leads to better performances than SIMON for round-based ASIC implementations, arguably the most important type of implementation since it provides a very good throughput for a reasonably low area cost, in contrary to serial implementations that only minimizes area. We also exhibit ASIC threshold implementations of our SKINNY variants that compare for example very favourably to AES-128 threshold implementations. As explained above, this is an integral part of modern lightweight primitives.

Regarding software, our implementations outperform all lightweight ciphers, except SIMON which performs slightly faster in the situation where the key schedule is performed only once. However, as remarked in [3], it is more likely in practice that the key schedule has to be performed everytime, and since SKINNY has a very lightweight key schedule we expect the efficiency of SKINNY software implementations to be equivalent to that of SIMON. This shows that SKINNY would perfectly fit a scenario where a server communicate with many lightweight devices. These performances are not surprising, in particular for bit-sliced implementations, as we show that SKINNY uses a much smaller total number of AND/NOR/XOR gates compared to all known lightweight block ciphers. This indicates that SKINNY will be competitive for most platforms and scenarios. Micro-controllers are no exception, and we show that SKINNY performs extremely well on these architectures.

We further remark that the decryption process of SKINNY has almost exactly the same description as the encryption counterpart, thus minimizing the decryption overhead.

We finally note that similarly to SIMON, SKINNY very naturally encompasses 64- or 128-bit block versions and a wide range of key sizes. However, in addition, SKINNY provides a tweakable capability, which can be very useful not only for leakage resilient implementations, but also to be directly plugged into higher-level operating modes, such as SCT [16]. In order to provide this tweak feature, we have generalized the STK construction [10] to enable more compact implementations while maintaining a high provable security level.

2 Specifications of SKINNY

Notations and SKINNY Versions. The lightweight block ciphers of the SKINNY family have 64-bit and 128-bit block versions and we denote $n$ the block size. In both $n = 64$ and $n = 128$ versions, the internal state is viewed as a $4 \times 4$ square array of cells, where each cell is a nibble (in the $n = 64$ case) or a byte (in the $n = 128$ case). We denote $IS_{i,j}$ the cell of the internal state located at Row $i$ and Column $j$ (counting starting from 0). One can also view this $4 \times 4$ square array of cells as a vector of cells by concatenating the rows. Thus, we denote with a single subscript $IS_i$ the cell of the internal state located at Position $i$ in this vector (counting starting from 0) and we have that $IS_{i,j} = IS_{4i+j}$.

SKINNY follows the TWEAKEY framework from [10] and thus takes a tweak input instead of a key or a pair key/tweak. The user can then choose what part of this tweak input will be key material and/or tweak material (classical block cipher view is to use the entire tweak input as key material only). The family of lightweight block ciphers SKINNY have three main tweakable size versions: for a block size $n$, we propose versions with tweaksize $t = n$, $t = 2n$ and $t = 3n$ (versions with other tweak sizes between $n$ and $3n$ are naturally obtained from these main versions) and we denote $z = t/n$ the tweak...
size to block size ratio. The tweakey state is also viewed as a collection of \( z \times 4 \times 4 \) square arrays of cells of \( s \) bits each. We denote these arrays \( TK1 \) when \( z = 1 \), \( TK1 \) and \( TK2 \) when \( z = 2 \), and finally \( TK1 \), \( TK2 \) and \( TK3 \) when \( z = 3 \). Moreover, we denote \( TK_{z,i,j} \) the cell of the tweakey state located at Row \( i \) and Column \( j \) of the \( z \)-th cell array. As for the internal state, we extend this notation to a vector view with a single subscript: \( TK_1, TK_2 \), and \( TK_3 \). Moreover, we define the adversarial model \( \text{SK} \) (resp. \( TK_1, TK_2 \) or \( TK_3 \)) where the attacker cannot (resp. can) introduce differences in the tweakey state.

**Initialization.** The cipher receives a plaintext \( m = m_0 \| m_1 \| \cdots \| m_{14} \| m_{15} \), where the \( m_i \) are \( s \)-bit cells, with \( s = n/16 \) (we have \( s = 4 \) for the 64-bit block \( \text{SKINNY} \) versions and \( s = 8 \) for the 128-bit block \( \text{SKINNY} \) versions). The initialization of the cipher’s internal state is performed by simply setting \( IS_i = m_i \) for \( 0 \leq i \leq 15 \):

\[
IS = \begin{bmatrix}
m_0 & m_1 & m_2 & m_3 \\
m_4 & m_5 & m_6 & m_7 \\
m_8 & m_9 & m_{10} & m_{11} \\
m_{12} & m_{13} & m_{14} & m_{15}
\end{bmatrix}
\]

This is the initial value of the cipher internal state and note that the state is loaded row-wise rather than in the column-wise fashion we have come to expect from the \( \text{AES} \); this is a more hardware-friendly choice, as pointed out in [14].

The cipher receives a tweakey input \( tk = tk_0 \| tk_1 \| \cdots \| tk_{16} \| tk_{16+1} \), where the \( tk_i \) are \( s \)-bit cells. The initialization of the cipher’s tweakey state is performed by simply setting for \( 0 \leq i \leq 15 \): \( TK1_i = tk_i \) when \( z = 1 \), \( TK2_i = tk_i \) when \( z = 2 \), and finally \( TK3_i = tk_i \) when \( z = 3 \). We note that the tweakey states are loaded row-wise.

**Table 1.** Number of rounds for \( \text{SKINNY}-n-t \), with \( n \)-bit internal state and \( t \)-bit tweakey state.

<table>
<thead>
<tr>
<th>Block size ( n )</th>
<th>( n )</th>
<th>( 2n )</th>
<th>( 3n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>32</td>
<td>36</td>
<td>40</td>
</tr>
<tr>
<td>128</td>
<td>40</td>
<td>48</td>
<td>56</td>
</tr>
</tbody>
</table>

**The Round Function.** One encryption round of \( \text{SKINNY} \) is composed of five operations in the following order: \( \text{SubCells}, \text{AddConstants}, \text{AddRoundTweakey}, \text{ShiftRows} \) and \( \text{MixColumns} \) (see illustration in Figure 1). The number \( r \) of rounds to perform during encryption depends on the block and tweakey sizes. The actual values are summarized in Table 1. Note that no whitening key is used in \( \text{SKINNY} \). Thus, a part of the first and last round do not add any security. We motivate this choice in Section 3.
SubCells. A $s$-bit Sbox is applied to every cell of the cipher internal state. For $s = 4$, SKINNY cipher uses a Sbox $S_4$ very close to the PICCOLO Sbox [19]. The action of this Sbox in hexadecimal notation is given by the following Table 2.

<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>$S_4[x]$</td>
<td>c</td>
<td>6</td>
<td>9</td>
<td>0</td>
<td>1</td>
<td>a</td>
<td>2</td>
<td>b</td>
<td>3</td>
<td>8</td>
<td>5</td>
<td>d</td>
<td>4</td>
<td>e</td>
<td>7</td>
<td>f</td>
</tr>
<tr>
<td>$S_4^{-1}[x]$</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>c</td>
<td>a</td>
<td>1</td>
<td>e</td>
<td>9</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td>0</td>
<td>b</td>
<td>d</td>
<td>f</td>
</tr>
</tbody>
</table>

Note that $S_4$ can also be described with four NOR and four XOR operations, as depicted in Figure 2. If $x_0, x_1, x_2$ and $x_3$ represent the four inputs bits of the Sbox ($x_0$ being the least significant bit), one simply applies the following transformation:

$$(x_3, x_2, x_1, x_0) \rightarrow (x_3, x_2, x_1, x_0 \oplus (x_3 \lor x_2)),$$

followed by a left shift bit rotation. This process is repeated four times, except for the last iteration where the bit rotation is omitted.

[Fig. 2. Construction of the Sbox $S_4$.]

[Fig. 3. Construction of the Sbox $S_8$.]

For the case $s = 8$, SKINNY uses an 8-bit Sbox $S_8$ that is built in a similar manner as for the 4-bit Sbox $S_4$ described above. The construction is simple and is depicted in Figure 3. If $x_0, \ldots, x_7$ represent the eight inputs bits of the Sbox ($x_0$ being the least significant bit), it basically applies the below transformation on the 8-bit state:

$$(x_7, x_6, x_5, x_4, x_3, x_2, x_1, x_0) \rightarrow (x_7, x_6, x_5, x_4 \oplus (x_7 \lor x_6), x_3, x_2, x_1, x_0 \oplus (x_3 \lor x_2)),$$

followed by the bit permutation:

$$(x_7, x_6, x_5, x_4, x_3, x_2, x_1, x_0) \rightarrow (x_2, x_1, x_7, x_6, x_4, x_0, x_3, x_5),$$

repeating this process four times, except for the last iteration where there is just a bit swap between $x_1$ and $x_2$. Besides, we provide in Appendix A the table of Sbox $S_8$ and its inverse in hexadecimal notations.
**AddConstants.** A 6-bit affine LFSR, whose state is denoted \((rc_5, rc_4, rc_3, rc_2, rc_1, rc_0)\) (with \(rc_0\) being the least significant bit), is used to generate round constants. Its update function is defined as:

\[
(rc_5||rc_4||rc_3||rc_2||rc_1||rc_0) \rightarrow (rc_4||rc_3||rc_2||rc_1||rc_0||rc_5 \oplus rc_4 \oplus 1).
\]

The six bits are initialized to zero, and updated before use in a given round. The bits from the LFSR are arranged into a \(4 \times 4\) array (only the first column of the state is affected by the LFSR bits), depending on the size of internal state:

\[
\begin{bmatrix}
  c_0 & 0 & 0 & 0 \\
  c_1 & 0 & 0 & 0 \\
  c_2 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 \\
\end{bmatrix},
\]

with \(c_2 = 0x2\) and

\[
(c_0, c_1) = (rc_5||rc_2||rc_1||rc_0, 0||0||rc_5||rc_4) \text{ when } s = 4,
\]

\[
(c_0, c_1) = (0||0||0||0||0||0||0||0||0||0||rc_5||rc_4) \text{ when } s = 8.
\]

The round constants are combined with the state, respecting array positioning, using bitwise exclusive-or. The values of the \((rc_5, rc_4, rc_3, rc_2, rc_1, rc_0)\) constants for each round are given in the table below, encoded to byte values for each round, with \(rc_0\) being the least significant bit.

<table>
<thead>
<tr>
<th>Rounds</th>
<th>Constants</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 - 16</td>
<td>01, 03, 07, 0F, 1F, 3E, 3D, 3B, 37, 2F, 1E, 3C, 39, 33, 27, 0E</td>
</tr>
<tr>
<td>17 - 32</td>
<td>1D, 3A, 35, 2B, 16, 2C, 1B, 80, 02, 05, 0B, 17, 2E, 1C, 38</td>
</tr>
<tr>
<td>33 - 48</td>
<td>31, 23, 06, 0D, 1B, 36, 2D, 1A, 34, 29, 12, 24, 08, 11, 22, 04</td>
</tr>
<tr>
<td>49 - 62</td>
<td>09, 13, 26, 0C, 19, 32, 25, 0A, 15, 2A, 14, 28, 10, 20</td>
</tr>
</tbody>
</table>

**AddRoundTweakey.** The first and second rows of all tweakey arrays are extracted and bitwise exclusive-or to the cipher internal state, respecting the array positioning. More formally, for \(i = \{0, 1\}\) and \(j = \{0, 1, 2, 3\}\), we have:

- \(IS_{i,j} = IS_{i,j} \oplus TK1_{i,j}\) when \(z = 1\),
- \(IS_{i,j} = IS_{i,j} \oplus TK1_{i,j} \oplus TK2_{i,j}\) when \(z = 2\),
- \(IS_{i,j} = IS_{i,j} \oplus TK1_{i,j} \oplus TK2_{i,j} \oplus TK3_{i,j}\) when \(z = 3\).

![Fig. 4. The tweakey schedule in SKINNY. Each tweakey word TK1, TK2 and TK3 (if any) follows a similar transformation update, except that no LFSR is applied to TK1.](image)

6
Then, the tweakkey arrays are updated as follows (this tweakkey schedule is illustrated in Figure 4). First, a permutation \( P_T \) is applied on the cells positions of all tweakkey arrays: for all \( 0 \leq i \leq 15 \), we set \( TK_1 \leftarrow TK_1 P_T[i] \) with
\[
P_T = [9, 15, 8, 13, 10, 14, 12, 11, 0, 1, 2, 3, 4, 5, 6, 7],
\]
and similarly for \( TK_2 \) when \( z = 2 \), and for \( TK_2 \) and \( TK_3 \) when \( z = 3 \). This corresponds to the following reordering of the matrix cells, where indices are taken row-wise:
\[
(0, \ldots, 15) \xmapsto{\cdot P_T} (9, 15, 8, 13, 10, 14, 12, 11, 0, 1, 2, 3, 4, 5, 6, 7)
\]
Finally, every cell of the first and second rows of \( TK_2 \) and \( TK_3 \) (for the SKINNY versions where \( TK_2 \) and \( TK_3 \) are used) are individually updated with an LFSR. The LFSRs used are given in Table 3 (\( x_0 \) stands for the LSB of the cell).

<table>
<thead>
<tr>
<th>TK</th>
<th>( s )</th>
<th>LFSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>( TK_2 )</td>
<td>4</td>
<td>( (x_3</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>( (x_7</td>
</tr>
<tr>
<td>( TK_3 )</td>
<td>4</td>
<td>( (x_3</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>( (x_7</td>
</tr>
</tbody>
</table>

**ShiftRows.** As in AES, in this layer the rows of the cipher state cell array are rotated, but they are to the right. More precisely, the second, third, and fourth cell rows are rotated by 1, 2 and 3 positions to the right, respectively. In other words, a permutation \( P \) is applied on the cells positions of the cipher internal state cell array: for all \( 0 \leq i \leq 15 \), we set \( IS_i \leftarrow IS_P[i] \) with
\[
P = [0, 1, 2, 3, 4, 5, 6, 10, 11, 8, 9, 13, 14, 15, 12].
\]

**MixColumns.** Each column of the cipher internal state array is multiplied by the following binary matrix \( M \):
\[
M = \begin{pmatrix}
1 & 0 & 1 & 1 \\
1 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 \\
1 & 0 & 1 & 0
\end{pmatrix}.
\]

The final value of the internal state array provides the ciphertext with cells being unpacked in the same way as the packing during initialization. Test vectors for SKINNY are provided in Appendix B. Note that decryption is very similar to encryption as all cipher components have very simple inverse (SubCells and MixColumns are based on a generalized Feistel structure, so their respective inverse is straightforward to deduce and can be implemented with the exact same number of operations).

**Extending to Other Tweakey Sizes.** The three main versions of SKINNY have tweakkey sizes \( t = n \), \( t = 2n \) and \( t = 3n \), but one can easily extend this to any size\(^2\) of tweakkey \( n \leq t \leq 3n \):

\(^2\)For simplicity we do not include here tweakkey sizes that are not a multiple of \( s \) bits. However, such cases can be trivially handled by generalizing the tweakkey schedule description to the bit level.
• for any tweakey size $n < t < 2n$, one simply uses exactly the $t = 2n$ version but the last $2n - t$ bits of the tweakey state are fixed to the zero value. Moreover, the corresponding cells in the tweakey state $TK2$ will not be updated throughout the rounds with the LFSR.

• for any tweakey size $2n < t < 3n$, one simply uses exactly the $t = 3n$ version but the last $3n - t$ bits of the tweakey state are fixed to the zero value. Moreover, the corresponding cells in the tweakey state $TK3$ will not be updated throughout the rounds with the LFSR.

We note that some of our 64-bit block SKINNY versions allow small key sizes (down to 64-bit). We emphasize that we propose these versions mainly for simplicity in the description of the SKINNY family of ciphers. Yet, as advised by the NIST [15], one should not to use key sizes that are smaller than 112 bits.

**Instantiating the Tweakey State with Key and Tweak Material.** Following the TWEAKEY framework [10], SKINNY takes as inputs a plaintext or a ciphertext and a tweakey value, which can be used in a flexible way by filling it with key and tweak material. Whatever the situation, the user must ensure that the key size is always at least as big as the block size.

In the classical setting where only key material is input, we use exactly the specifications of SKINNY described previously. However, when some tweak material is to be used in the tweakey state, we dedicate $TK1$ for this purpose and XOR a bit set to “1” every round to the second bit of the top cell of the third column (i.e., the second bit of $IS_{0,2}$). In other words, when there is some tweak material, we add an extra “1” in the constant matrix from AddConstants. Besides, in situations where the user might use different tweak sizes, we recommend to dedicate some cells of $TK1$ to encode the size of the tweak material, in order to ensure proper separation. Note that these are only recommendations, thus not strictly part of the specifications of SKINNY.

### 3 Rationale of SKINNY

Several design choices of SKINNY have been borrowed from existing ciphers, but most of our components are new, optimized for our goal: a cipher well suited for most lightweight applications. When designing SKINNY, one of our main criteria was to only add components which are vital for the security of the primitive, removing any unnecessary operation (hence the name of our proposal). We end up with the sound property that removing any component or using weaker version of a component from SKINNY would lead to a much weaker (or actually insecure) cipher. Therefore, the construction of SKINNY has been done through several iterations, trying to reach the exact spot where good performance meets strong security arguments. We detail in this section how we tried to follow this direction for each layer of the cipher.

We note that one could have chosen a slightly smaller Sbox or a slightly sparser diffusion layer, but our preliminary implementations showed that these options represent worse tradeoff overall. For example, one could imagine a very simple cipher iterating thousands of rounds composed of only a single non-linear boolean operation, an XOR and some bit wiring. However, such a cipher will lead to terrible performance regarding throughput, latency or energy consumption.

When designing a lightweight encryption scheme, several use cases must be taken in account. While area optimized implementations are important for some very constrained applications, throughput or throughput-over-area optimized implementations are also very relevant. Actually, looking at recently introduced efficiency measurements [12], one can see that our designs choices are good for many types of implementations, which is exactly what makes a good general-purpose lightweight encryption scheme.

#### 3.1 General Design and Components Rationale

A first and important decision was to choose between a Substitution-Permutation Network (SPN), or a Feistel network. We started from a SPN construction as it is generally easier to provide stronger bounds on the number of active Sboxes. However, we note that there is a dual bit-sliced view of SKINNY that resembles some generalized Feistel network. Somehow, one can view the cipher as a primitive in between...
an SPN and an “AND-rotation-XOR” function like SIMON. We try to get the best of both worlds by benefiting the nice implementation tradeoffs of the latter, while organizing the state in an SPN view so that bounds on the number of active Sboxes can be easily obtained.

The absence of whitening key is justified by the reduction of the control logic: by always keeping the exact same round during the entire encryption process we avoid the control logic induced by having a last non-repeating layer at the end of the cipher. Besides, this simplifies the general description and implementation of the primitive. Obviously, having no whitening key means that a few operations of the cipher have no impact on the security. This is actually the case for both the beginning and the end of the ciphering process in SKINNY since the key addition is done in the middle of the round, with only half of the state being involved with this key addition every round.

A crucial feature of SKINNY is the easy generation of several block size or tweak size versions, while keeping the general structure and most of the security analysis untouched. Going from the 64-bit block size versions to the 128-bit block size versions is simply done by using a 8-bit Sbox instead of a 4-bit Sbox, therefore keeping all the structural analysis identical. Using bigger tweakkey material is done by following the STRK construction [10], which allows automated analysis tools to still work even though the input space become very big (in short, the superposition trick makes the TK2 and TK3 analysis almost as time consuming as the normal and easy TK1 case). Besides, unlike previous lightweight block ciphers, this complete analysis of the TK2 and TK3 cases allows us to dedicate a part of this tweakkey material to be potentially some tweak input, therefore making SKINNY a flexible tweakable block cipher. Also, we directly obtain related-key security proofs using this general structure.

SubCells. The choice of the Sbox is obviously a crucial decision in an SPN cipher and we have spent a lot of efforts on looking for the best possible candidate. For the 4-bit case, we have designed a tool that searches for the most compact candidate that provides some minimal security guarantees. Namely, with the bit operations cost estimations given previously, for all possible combinations of operations (NAND/NOR/XOR/XNOR) up to a certain limit cost, our tool checks if certain security criterion of the tested Sbox are fulfilled. More precisely, we have forced the maximal differential transition probability of the Sbox to be $2^{-2}$ and the maximal absolute linear bias to be $2^{-2}$. When both criteria are satisfied, we have filtered our search for Sbox with high algebraic degree.

Our results is that the Sbox used in the PICCOLO block cipher [19] is close to be the best one: our 4-bit Sbox candidate $S_4$ is essentially the PICCOLO Sbox with the last NOT gate at the end being removed (see Figure 2). We believe this extra NOT gate was added by the PICCOLO designers to avoid fixed points (actually, if fixed points were to be removed at the Sbox level, the PICCOLO candidate would be the best choice), but in SKINNY the fixed points are handled with the use of constants to save some extra GE. Yet, omitting the last bit rotation layer removes already a lot of fixed points (the efficiency cost of this omission being null).

The Sbox $S_4$ can therefore be implemented with only 4 NOR gates and 4 XOR gates, the rest being only bit wiring (basically free in hardware). According to our previously explained estimations, this should cost 14.68 GE, but as remarked in [19], some libraries provide special gates that further save area. Namely, in our library the 4-input AND-NOR and 4-input OR-NAND gates with two inputs inverted cost 2 GE and they can be used to directly compute a XOR or an XNOR. Thus, $S_4$ can be implemented with only 12 GE. In comparison, the PRESENT Sbox [4] requires 3 AND, 1 OR and 11 XOR gates, which amounts to 27.32 GE (or 34.69 GE without the special 4-input gates).

All in all, our 4-bit Sbox $S_4$ has the following security properties: maximal differential transition probability of $2^{-2}$, maximal absolute linear bias of $2^{-2}$, branching number 2, algebraic degree 3 and one fixed point $S_4(0xF) = 0xF$.

Regarding the 8-bit Sbox, the search space was too wide for our automated tool. Therefore, we instead considered a subclass of the entire search space: by reusing the general structure of $S_4$, we have tested all possible Sboxes built by iterating several times a NOR/XOR combination and a bit permutation. Our search found that the maximal differential transition probability and maximal absolute linear bias of the Sboxes are larger than $2^{-2}$ when we have less than 8 iterations of the NOR/XOR combination and bit permutation. With 8 iterations of the NOR/XOR combination and bit permutation, we found Sboxes
with desired maximal differential transition probability of $2^{-2}$ and maximal absolute linear bias of $2^{-2}$ with algebraic degree 6. However, the algebraic degree of the inverse Sboxes of all these candidates is 5 rather than 6. In addition, having 8 iterations may result in higher latency when we consider a serial hardware implementation. Therefore, we considered having 2 NOR/XOR combinations in every iteration and reduce the number of iteration from 8 to 4. As a result, we found several Sboxes with the desired maximal differential probability and absolute linear bias, while reaching algebraic degree 6 for both the Sbox and its inverse (thus better than the 8 iterations case). Although such Sbox candidates have 3 fixed points when we omit the last bit permutation layer like the 4-bit case, we can easily reduce the number of fixed points by introducing a different bit permutation from the intermediate bit permutations to the last layer without any additional cost.

With 2 NOR/XOR combinations and a bit permutation iterated 4 times, $S_6$ can be implemented with only 8 NOR gates and 8 XOR gates (see Figure 3), the rest being only bit wiring (basically free in hardware). The total area cost should be 24 GE according to our previously explained estimations and using special 4-input AND-NOR and 4-input OR-NAND gates. In comparison, while ensuring a maximal differential transition probability (resp. maximum absolute linear bias) of $2^{-6}$ (resp. $2^{-4}$), the AES Sbox requires 32 AND/OR gates and 83 XOR gates to be implemented, which amounts to 198 GE. Even recent lightweight 8-bit Sbox proposal [6] requires 12 AND/OR gates and 26 XOR gates, which amounts to 64 GE, for a maximal differential transition probability (resp. maximum linear bias) of $2^{-5}$ (resp. $2^{-2}$), but their optimization goal was different from ours.

All in all, we believe our 8-bit Sbox candidate $S_6$ provides a good tradeoff between security and area cost. It has maximal differential transition probability of $2^{-2}$, maximal absolute linear bias of $2^{-2}$, branching number 2, algebraic degree 6 and a single fixed point $S_6(0xFF) = 0xFF$ (for the Sbox we have chosen, swapping two bits in the last bit permutation was probably the simplest method to achieve only a single fixed point).

Note that both our Sboxes $S_4$ and $S_6$ have the interesting feature that their inverse is computed almost identically to the forward direction (as they are based on a generalized Feistel structure) and with exactly the same number of operations. Thus, our design reasoning also holds when considering the decryption process.

**AddConstants.** The constants in SKINNY have several goals: differentiate the rounds, differentiate the columns and avoid symmetries, complicate subspace cryptanalysis and attacks exploiting fixed points from the Sbox. In order to differentiate the rounds, we simply need a counter, and since the number of rounds of all SKINNY versions is smaller than 64, the most hardware friendly solution is to use a very cheap 6-bit affine LFSR (like in LED [9]) that requires only a single XNOR gate per update. The 6 bits are then dispatched to the two first rows of the first column (this will maximize the constants spread after the ShiftRows and MixColumns), which will already break the columns symmetry.

In order to avoid symmetries, fixed points and more generally subspaces to spread, we need to introduce different constants in several cells of the internal state. The round counter will already naturally have this goal, yet, in order to increase that effect, we have added a “1” bit to the third row, which is almost free in terms of implementation cost. This will ensure that symmetries and subspaces are broken even more quickly, and in particular independently of the round counter.

**AddRoundTweakey.** The tweakey schedule of SKINNY follows closely the STK construction from [10] (that allows to easily get bounds on the number of active Sboxes in the related-tweakey model). Yet, we have changed a few parts. Firstly, instead of using multiplications by 2 and 3 in a finite field, we have instead replaced these tweakey cells updates by cheap 4-bit or 8-bit LFSRs (depending on the size of the cell) to minimize the hardware cost. All our LFSRs require only a single XOR for the update, and we have checked that the differential cancellation behavior of these interconnected LFSRs is as required by the STK construction: for a given position, a single cancellation can only happen every 15 rounds for TK2, and same with two cancellations for TK3.

Another important generalization of the STK construction is the fact that every round we XOR only half of the internal cipher state with some subtweakey. The goal was clearly to optimize hardware performances.
of SKINNY, and it actually saves an important amount of XORs in a round-based implementation. The potential danger is that the bounds we obtain would dramatically drop because of this change. Yet, surprisingly, the bounds remained actually good and this was a good security/performance tradeoff to make. Another advantage is that we can now update the tweak key cells only before they are incorporated to the cipher internal state. Thus, half of tweak key cells only will be updated every round and the period of the cancellations naturally doubles: for a certain cell position, a single cancellation can only happen every 30 rounds for TK2 and two cancellations can only happen every 30 rounds for TK3.

The tweak permutation $P_T$ has been chosen to maximize the bounds on the number of active Sboxes that we could obtain in the related-tweak key model (note that it has no impact in the single-key model). Besides, we have enforced for $P_T$ the special property that all cells located in third and fourth rows are sent to the first and second rows, and vice-versa. Since only the first and second rows of the tweak key states are XORed to the internal state of the cipher, this ensures that both halves of the tweak key states will be equally mixed to the cipher internal state (otherwise, some tweak key bytes might be more involved in the ciphering process than others). Finally, the cells that will not be directly XORed to the cipher internal state can be left at the same relative position. On top of that, we only considered those variants of $P_T$ that consist of a single cycle.

We note that since the cells of the first tweak key word $TK1$ are never updated, they can be directly hardwired to save some area if the situation allows.

ShiftRows and MixColumns. Competing with SIMON’s impressive hardware performance required choosing an extremely sparse diffusion layer for SKINNY, which was in direct contradiction with our original goal of obtaining good security bounds for our primitive. Note that since our Sboxes $S_i$ and $S_k$ have a branching number of two, we cannot use only a bit permutation layer as in the PRESENT block cipher: differential characteristics with only a single active Sbox per round would exist. After several design iterations, we came to the conclusion that binary matrices were the best choice. More surprisingly, while most block cipher designs are using very strong diffusion layers (like an MDS matrix), and even though a $4 \times 4$ binary matrices with branching number four exist, we preferred a much sparser candidate which we believe offers the best security/performance tradeoff (this can be measured in terms of Figure Of Adversarial Merit [12]).

Due to its strong sparseness, SKINNY binary diffusion matrix $M$ has only a differential or linear branching number of two. This seems to be worrisome as it would again mean that differential characteristics with only a single active Sbox per round would exist (it would be the same for PRESENT block cipher if its Sbox did not have branching number three, which is the reason of the relatively high cost of the PRESENT Sbox). However, we designed $M$ such that when a branching two differential transition occurs, the next round will likely lead to a much higher branching number. Looking at $M$, the only way to meet branching two is to have an input difference in either the second or the fourth input only. This leads to an input difference in the first or third element for the next round, which then diffuses to many output elements. The differential characteristic with a single active Sbox per round is therefore impossible, and actually we will be able to prove at least 96 active Sboxes for 20 rounds. Thus, for the very cheap price of a differential branching two binary diffusion matrix, we are in fact getting a better security than expected when looking at the iteration of several rounds. The effect is the same with linear branching (for which we only need to look at the transpose of the inverse of $M$, i.e. $(M^{-1})^\top$).

We have considered all possibilities for $M$ that can be implemented with at most three XOR operations and eventually kept the MixColumns matrices that, in combination with ShiftRows, guaranteed high diffusion and led to strong bounds on the minimal number of active Sboxes in the single-key model.

Note that another important criterion came into play regarding the choice of the diffusion layer of SKINNY: it is important that the key material impacts as fast as possible the cipher internal state. This is in particular a crucial point for SKINNY as only half of the state is mixed with some key material every round, and since there is no whitening key. Besides, having a fast key diffusion will reduce the impact of meet-in-the-middle attacks. Once the two first rows of the state were arbitrarily chosen to receive the key material, given a certain subkey, we could check how many rounds were required (in both encryption and decryption directions) to ensure that the entire cipher state depends on this
subtweakey. Our final choice of MixColumns is optimal: only a single round is required in both forward and backward directions to ensure this diffusion.

4 Implementations, Performance and Comparison

4.1 ASIC Round-Based Implementations

This section is dedicated to the description of the different hardware implementations of all variants of SKINNY. We used Synopsys DesignCompiler version A-2007.12-SP1 to synthesize the designs considering UMCL18G212T3 [22] standard cell library, which is based on the UMC L180 0.18µm 1P6M logic process with a typical voltage of 1.8 V. For the synthesis, we advised the compiler to keep the hierarchy and use a clock frequency of 100 KHz, which allows a fair comparison with the benchmark of other block ciphers reported in literature.

We designed round-based implementations for all SKINNY variants providing a good trade-off between performance and area. All implementations compute a single round of SKINNY within a clock cycle. Besides, our designs take advantage of dedicated scan flip-flops rather than using simple flip-flops and additional multiplexers placed in front in order to hold round states and keys. Note that this approach leads to savings of 1 GE per bit to be stored. In order to allow a better and fairer comparison, we provide both throughput at a maximally achievable frequency and throughput at a frequency of 100KHz.

<table>
<thead>
<tr>
<th>Area</th>
<th>Delay</th>
<th>Clock</th>
<th>Throughput</th>
<th>Ref.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GE</td>
<td>ns</td>
<td>Cycles @100KHz</td>
<td>@maximum</td>
</tr>
<tr>
<td>SKINNY-64-64</td>
<td>1223</td>
<td>1.77</td>
<td>32</td>
<td>200.00</td>
</tr>
<tr>
<td>SKINNY-64-128</td>
<td>1696</td>
<td>1.87</td>
<td>36</td>
<td>177.78</td>
</tr>
<tr>
<td>SKINNY-64-192</td>
<td>2183</td>
<td>2.02</td>
<td>40</td>
<td>160.00</td>
</tr>
<tr>
<td>SKINNY-128-128</td>
<td>2391</td>
<td>2.89</td>
<td>40</td>
<td>320.00</td>
</tr>
<tr>
<td>SKINNY-128-256</td>
<td>3312</td>
<td>2.89</td>
<td>48</td>
<td>266.67</td>
</tr>
<tr>
<td>SKINNY-128-384</td>
<td>4268</td>
<td>2.89</td>
<td>56</td>
<td>228.57</td>
</tr>
<tr>
<td>SIMON-64-128</td>
<td>1751</td>
<td>1.60</td>
<td>46</td>
<td>145.45</td>
</tr>
<tr>
<td>SIMON-128-128</td>
<td>2342</td>
<td>1.60</td>
<td>70</td>
<td>188.24</td>
</tr>
<tr>
<td>SIMON-128-256</td>
<td>3419</td>
<td>1.60</td>
<td>74</td>
<td>177.78</td>
</tr>
<tr>
<td>LED-64-64</td>
<td>2695</td>
<td>-</td>
<td>32</td>
<td>198.90</td>
</tr>
<tr>
<td>LED-64-128</td>
<td>3036</td>
<td>-</td>
<td>48</td>
<td>133.00</td>
</tr>
<tr>
<td>PRESENT-64-128</td>
<td>1884</td>
<td>-</td>
<td>32</td>
<td>200.00</td>
</tr>
<tr>
<td>PICCOLO-64-128</td>
<td>1773</td>
<td>-</td>
<td>33</td>
<td>193.94</td>
</tr>
</tbody>
</table>

1 This number includes 576 GE for key storage that is not considered in the original work.

4.2 Software Implementations

In this section, we detail how the ciphers in the SKINNY family can be implemented in software. More precisely, we consider four of the latest Intel processors using SIMD instruction sets to perform efficient parallel computations of several input blocks. We give in particular the performance figures for a bit-sliced implementations of SKINNY.
Bit-Sliced Implementations of SKINNY. Since the design of SKINNY has been made with hardware implementations in mind, the conversion to bit-sliced implementations seems natural. In the following, we target different sets of instructions, namely SSE4 and AVX2, which provide shuffling instructions on byte level, as well as several wide 128-bit resp. 256-bit registers, commonly referred as XMM or YMM registers. From our perspective, the main differences between SSE4 and AVX2 are the width of the available registers and the possibility to use 3-operand instructions.

In the Table 5, we give the detailed performance figures of our implementations in the case of SKINNY-64 and compare it with other ciphers. Note that these implementations take into account all data transformations which are required. The bit-sliced implementations for SIMON processing 32 resp. 64 blocks have been provided by the designers to allow us a fair comparison in the same setting.

Table 5. Bit-sliced implementations of SKINNY-64, SKINNY-128 and other 64-bit block lightweight ciphers. Performances are given in cycles per byte, with pre-expanded subkeys. For SKINNY-64 and SIMON encryption we encrypted 2000 64-bit blocks to obtain the results. Cells with dashes (-) represent non-existing implementations to date.

<table>
<thead>
<tr>
<th>Parallelization</th>
<th>Haswell 16</th>
<th>Haswell 32</th>
<th>Haswell 64</th>
<th>Skylake 16</th>
<th>Skylake 32</th>
<th>Skylake 64</th>
<th>Ref.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SKINNY-128-128</td>
<td>-</td>
<td>-</td>
<td>4.32</td>
<td>-</td>
<td>-</td>
<td>3.96</td>
<td>New</td>
</tr>
<tr>
<td>SKINNY-64-128</td>
<td>-</td>
<td>-</td>
<td>3.05</td>
<td>-</td>
<td>-</td>
<td>2.78</td>
<td>New</td>
</tr>
<tr>
<td>SIMON-64-128</td>
<td>-</td>
<td>3.42</td>
<td>1.93</td>
<td>-</td>
<td>3.29</td>
<td>1.81</td>
<td></td>
</tr>
<tr>
<td>LED-128</td>
<td>22.6</td>
<td>13.7</td>
<td>-</td>
<td>23.1</td>
<td>13.3</td>
<td>-</td>
<td>[3]</td>
</tr>
<tr>
<td>PRESENT-128</td>
<td>10.8</td>
<td>-</td>
<td>-</td>
<td>10.3</td>
<td>-</td>
<td>-</td>
<td>[3]</td>
</tr>
<tr>
<td>Piccolo-128</td>
<td>9.2</td>
<td>-</td>
<td>-</td>
<td>9.2</td>
<td>-</td>
<td>-</td>
<td>[3]</td>
</tr>
</tbody>
</table>

Counter Mode Implementations of SKINNY-64-128. We also evaluate the speed of SKINNY-64-128 in the same conditions as the benchmarks provided in [1]. Namely, the goal is to generate the keystream from the counter mode using SKINNY-64-128 as the underlying cipher. The main difference to the previous scenario is that many blocks of a non-repeating value (counter) are encrypted. This allows to save the costs for data packing, as the values are known in advance and can already be provided in the correct format.

The designers of SIMON achieve a very high performances, by taking advantage of this mode, in their implementation available on GitHub. We would like to note that this CTR-mode implementation does not process the same amount of blocks as given in Table 5 and we expect the performance of SIMON to be closer to these figures for an optimized implementation.

In our case, we devise a very similar implementation that considers 64 blocks in parallel and reaches a maximal speed of 2.63 cpb in the same setting on the latest Intel platform Skylake. We note that the key is pre-expanded prior to encrypting the blocks, and the 64 blocks are stored in 16 registers of 256 bits in a bit-sliced way. In detail, the four first registers contain the four first bits of each first row of the 64 blocks. The same holds for the 12 others registers with the remaining three rows of the states.

Then, for all the 36 rounds of SKINNY-64-128, the application of SubCells, AddConstants, AddRoundTweakey, and MixColumns can be easily done with bit-wise operations on registers. As for ShiftRows, we implement it as a shuffle on bytes within each register. The benchmarks conducted on our four platforms are shown in Table 6.

---

3Available at https://github.com/lrwinge/simon_speck_supercop/.
Table 6. Counter mode implementations of SKINNY-64-128, SKINNY-128-128, SIMON-64-128 and SIMON-128-128. Performances are given in cycles per byte, with pre-expanded subkeys, encrypting 16384 bytes and obtained using SUPERCOP.

<table>
<thead>
<tr>
<th>Instruction Set</th>
<th>Westmere</th>
<th>Ivy Bridge</th>
<th>Haswell</th>
<th>Skylake</th>
<th>Ref.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sse4</td>
<td>sse4</td>
<td>sse4</td>
<td>sse4</td>
<td></td>
</tr>
<tr>
<td>SKINNY-64-128</td>
<td>7.87</td>
<td>5.27</td>
<td>5.14</td>
<td>2.92</td>
<td>4.79</td>
</tr>
<tr>
<td>SIMON-64-128</td>
<td>7.64</td>
<td>5.85</td>
<td>5.93</td>
<td>3.12</td>
<td>5.26</td>
</tr>
<tr>
<td>SKINNY-128-128</td>
<td>-</td>
<td>-</td>
<td>7.41</td>
<td>4.05</td>
<td>7.02</td>
</tr>
<tr>
<td>SIMON-128-128</td>
<td>11.52</td>
<td>8.87</td>
<td>8.70</td>
<td>4.39</td>
<td>7.99</td>
</tr>
</tbody>
</table>

References

A 8-bit Sbox for SKINNY-128

```c
/* SKINNY-128 Sbox */
uint8_t S[256] = {
0xe2,0xca,0x66,0x4e,0xe1,0xc8,0x32,0x88,
0x30,0x6c,0x85,0xc1,0x8d,0x07,0x12,0x83,
0x6e,0x0e,0x02,0x4c,0x9a,0xc0,0x09,0x3b,
0x2a,0x01,0x87,0x9f,0x4f,0x67,0x47,0x0f,
0x51,0x71,0x25,0x26,0x0f,0x37,0x77,0x0f,
0x1e,0xc8,0xec,0xc6,0xc4,0xc0,0x0d,0x8d,
0x36,0x8e,0x38,0xb2,0x8b,0x30,0x83,0x39,0x96,
0x26,0xa9,0x00,0x89,0x0b,0x83,0x06,0x76,
0x28,0x93,0x20,0x9b,0x2c,0x94,0x24,0x9d,
0x2a,0x02,0x8e,0x0f,0x00,0x33,0x0b,0x39,
0x26,0x46,0x24,0x2f,0x67,0x47,0x0f,0x51,
0x71,0x25,0x26,0x0f,0x37,0x77,0x0f,
};
```

B Test Vectors

The keys are given as the concatenation of (up to) three tweakwords: TK1, TK1||TK2, or TK1||TK2||TK3.

```c
/* SKINNY-64-64 */
Key: f5f269825fc6f8128
Plaintext: 06034f9f7724d19d
Ciphertext: bb39fd824299b8ac7
/* SKINNY-128 */
Key: 9eb93640d088da63
Plaintext: 76a396d1c8bea7f1e
Ciphertext: 6cede143d9e2b9e
/* SKINNY-64-128 */
Key: 6e00c8b5120d6861
Plaintext: 875e3a24bdf980f60
Ciphertext: 530c61d85e8663c3
/* SKINNY-64-192 */
Key: ed00c8b5120d6861
Plaintext: 875e3a24bdf980f60
Ciphertext: 530c61d85e8663c3
/* SKINNY-128-128 */
Key: 009ceccf1605d4ac1d2a9e3085df7a1f3
Plaintext: 2f0addbe6b8648a3b2ed1f0adda14
Ciphertext: 22ff3f0d498eae627e45b7e633675b74
/* SKINNY-128-256 */
Key: 0ac4776f726a084d382a69e7022e25
Plaintext: 7b319d9a4bde147a7ed4a6f1b9b587f
/* SKINNY-128-384 */
Key: df889548cf7ea52d296339301797449
Plaintext: ab588a34a471a1b2b4fc9283f8ea95
Ciphertext: 94ecf589e2017c601b38c6346a10dcfa
```

22. Virtual Silicon Inc.: 0.18 µm VIP Standard Cell Library Tape Out Ready, Part Number: UMC18G212T3, Process: UMC Logic 0.18 µm Generic II Technology: 0.18 µm (July 2004)