High Throughput in Slices: The Case of PRESENT, PRINCE and KATAN64 Ciphers

Kostas Papapagiannopoulos(✉)

Department of Digital Security, Radboud University Nijmegen,
Nijmegen, The Netherlands
kostaspap88@gmail.com

Abstract. This paper presents high-throughput assembly implementations of PRESENT, PRINCE and KATAN64 ciphers for the ATtiny family of AVR microcontrollers. We report new throughput records, achieving the speed of 2967 clock cycles per block encryption for PRESENT, 1803 cycles for PRINCE and 23671 cycles for KATAN64. In addition, we offer insight into the 'slicing' techniques used for high throughput and their application to lightweight cryptographic implementations. We also demonstrate the speed-memory tradeoff by constructing high-throughput implementations with large memory requirements.

Keywords: PRESENT · PRINCE · KATAN64 · AVR · ATtiny · High-speed assembly implementation

1 Introduction

During the recent years, our society experienced big changes in the IT landscape. Starting from the development of wireless connectivity and embedded systems, we have observed an extensive deployment of tiny computing devices in our environment. Everyday objects transform into sophisticated appliances, enhanced with communication and computation capabilities. Ubiquitous computing is gradually becoming a reality and researchers have already identified a wide range of security and privacy risks stemming from it.

In this new fully-interconnected, always-online environment, we rely heavily on a huge number of daily transactions that are carried over a large distributed infrastructure and can be security-critical or privacy-related. RFID tags on commercial products, cardiac pacemakers, fire-detecting sensor nodes, traffic jam detectors and vehicular ad-hoc communication systems have one thing in common: they need to establish a *secure* and *privacy-friendly* modus operandi, under a particularly restricted environment, *e.g.* limited processing capabilities, low energy consumption and/or demanding network protocols.

To provide sufficient security in such a setting, we need security primitives that have a small footprint (low gate number and construction complexity), reduced power consumption (since we often rely on a limited battery or on an external electromagnetic field to supply the required energy) and sufficient speed

N. Saxena and A.-R. Sadeghi (Eds.): RFIDSec 2014, LNCS 8651, pp. 137–155, 2014.
DOI: 10.1007/978-3-319-13066-8_9

(to be able to communicate in real time). The new pervasive computing requirements, in combination with the lack of a suitable candidate (AES is usually too expensive, despite various approaches that have been proposed to reduce the costs of hardware and software implementations [32]), has led researchers to establish new ciphers that are tailor-made for pervasive computing and are often referred to as lightweight ciphers. Among the best studied algorithms are the block ciphers CLEFIA [43], Hight [29], KATAN, KTAN-TAN [16], Klein [25], LED [27], PRESENT [11], the stream ciphers Grain [28], Mickey [7] and Trivium [17] and more recently lightweight hash functions such as SPONGENT [10], PHOTON [26] and QUARK [6].

Our contribution. This work focuses on the software speed aspect of lightweight cryptography, usually with CTR mode of encryption. For AVR devices with the ATtiny RISC architecture [20] (ATtiny85 and ATtiny45), we present new encryption throughput records for ciphers PRESENT and KATAN64 that improve the current state of the art ([21,39,41]) and we also present the first high-throughput implementation of PRINCE cipher. Our main tools for high-throughput are 'slicing' techniques, namely the traditional 'bitslicing' for PRESENT, a variant called 'nibble-slicing' for PRINCE and finally, hardware slices in KATAN64. We note that all these optimization techniques incur a large overhead in memory requirements. The ATtiny devices are low-power 8-bit AVR microcontrollers that employ SRAM, flash and EEPROM types of memory, as well as 32 registers, an ALU[1] and other peripherals. In Sects. 2, 3, 4 we explain these techniques and their effects in detail for PRESENT, PRINCE and KATAN64 respectively and provide comparisons between them. We measure directly the number of clock cycles, SRAM memory bytes and flash memory bytes that they require. We conclude in Sect. 5.

2 PRESENT Cipher: Bitslicing with 8-Bit Processors

This section of this work suggests a novel, bitsliced PRESENT cipher implementation that achieves high throughput performance, namely *2.9*× the throughput of the fastest non-bitsliced implementation (Papagiannopoulos, Verstegen [38,39]) and *2.1*× the throughput of the fastest bitsliced implementation to our knowledge (Rauzy, Guilley, Najm [41]). The second focal point of this section is to demonstrate the effects of 'slicing' techniques on cipher implementations. By opting for bitsliced PRESENT, we examine the speedups achieved in the permutation layer but also the repercussions occurring in the substitution layer under this non-standard, bitsliced representation.

Algorithm outline. PRESENT [11] is an ultra-lightweight, 64-bit symmetric block cipher, using 80-bit or 128-bit keys. It is based on a substitution/permutation network and as of 2012, was adopted as a standard for lightweight block ciphers (ISO/IEC 29192-2:2012 [3]). The full algorithm has been resistant to attempts at

[1] Arithmetic Logic Unit.

Table 1. Substitution layer. The S-box used in PRESENT is a 4-bit to 4-bit function S.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S[x]	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

Table 2. Permutation layer. The bit-oriented permutation network used in PRESENT. Bit in position i of state is moved to bit position $P(i)$.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P(i)	0	16	32	48	1	17	33	49	2	18	34	50	3	19	35	51
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
P(i)	4	20	36	52	5	21	37	53	6	22	38	54	7	23	39	55
i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
P(i)	8	24	40	56	9	25	41	57	10	26	42	58	11	27	43	59
i	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
P(i)	12	28	44	60	13	29	45	61	14	30	46	62	15	31	47	63

cryptanalysis, although attacks have shown that up to 15 of its 31 rounds can be broken with $2^{35.6}$ plaintext-ciphertext pairs in 2^{20} operations [4,18,36].

PRESENT uses exclusive-or as its round key operation, a 4-bit substitution layer, a bit permutation network with a 4-bit period, over 31 rounds and a final round key operation. Key scheduling is a combination of bit rotation, S-box application and exclusive-or with the round counter. Constructions found in PRESENT are also encountered in hash functions SPONGENT [10], H-PRESENT [12] and in ciphers Maya [24] and SMALL PRESENT [33]. Thus the optimizations presented here are also directly applicable to these algorithms or to any cipher that uses either a bit-oriented permutation network or the PRESENT S-box (*e.g.* the LED cipher [27]).

The cipher's key register is supplied with the 80-bit cipher key and in every encryption round the first 64 bits of the 80-bit key register form the round key. To encrypt a single 64-bit block, during each encryption round, PRESENT applies an exclusive-or with the current round key followed by a substitution and a permutation layer. The substitution layer applies nibble-wise (4-bit) S-boxes to the state (Table 1), while the permutation layer re-arranges the bits in the state following a 4-bit period (Table 2). Key scheduling is done by rotating the key register 61 bit positions to the left, applying the S-box to the top nibble of the key register and XORing bits 15 through 19 with the round counter. There is a total of 31 such rounds and finally we perform one last exclusive-or with the round key (Fig. 1).

2.1 Permutation Layer Under Bitslicing

Bitslicing was first introduced by Biham [8] in order to improve the performance of bit permutations of DES in software. We note that there exist structural

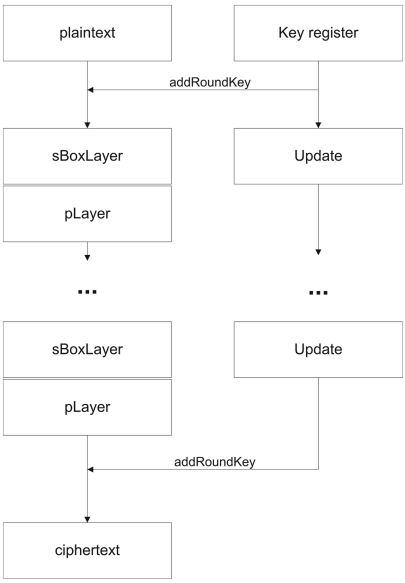

Fig. 1. Overview schematic of the PRESENT cipher. It consists of 31 rounds, including exclusive-or addRoundKey application, nibble-wise substitution (sBoxLayer), bit position permutation (pLayer) and key update.

similarities between DES and PRESENT; although DES is a Feistel instead of an SP network, both are hardware-oriented ciphers that rely heavily on bit permutations which are efficient with circuit wirings, yet slow in software. Bitslicing views our 8-bit microprocessor as a SIMD[2] computer with 8 single-bit processors running in parallel. Thus, we use a non-standard, bitsliced representation for our 64-bit PRESENT cipher block: 64 SRAM cells (each cell consisting of 8 bits) represent the 64 bit positions of the block. Due to the 8-bit size of our cells/positions, we are able to permute 8 cipher blocks in parallel. *i.e.* we achieve a bitslice factor of 8.

Normally, the permutation layer under this representation would be reduced to simple memory cell renaming according to the permutation pattern (Table 2) and should cost zero clock cycles. However, cell renaming for 31 cipher rounds requires full loop unrolling, resulting in infeasible code size. Thus, we use the following approach:

1. Load four 8-way-bitsliced cells from the SRAM to four registers.
2. Perform the nibble-based substitution layer (Sect. 2.2).
3. Store the substituted result back to SRAM cells in a permuted fashion (Table 2).
4. Repeat this for all nibbles in the cipher block.

[2] Single instruction, multiple data (SIMD), is a class of parallel computers in Flynn's taxonomy [23]. It describes computers with multiple processing elements that perform the same operation on multiple data points simultaneously.

Essentially, the computational cost of the permutation layer drops to 64 memory loads and 64 memory stores. In order to load cells in a sequential manner and to store in a permuted fashion, we use direct SRAM addressing (instructions `lds, sts`). The bitsliced permutation approach is substantially faster (in terms of throughput) when compared to LUT approaches like merged SP lookup tables [39] or permutation lookup tables [9]. Likewise, instruction-set-based approaches such as bit-level manipulation/masking techniques that employ the `bld, bst` instructions [37] or logical shifts [21] are also outperformed. The ineffectiveness of several ISAs[3] w.r.t. permutation operations has also been addressed by Lee *et al.* [34,42], who suggested extensions to existing instruction sets in order to compute arbitrary *n-bit* permutations.

2.2 Substitution Layer Under Bitslicing

Despite the large throughput boost on the permutation layer, bitslicing increases the complexity of the substitution operation and has even led to bitslicing-oriented compilers [40]. When assuming 4-bit S-boxes, a cipher block size of 64 bits and an 8-bit architecture, performing a substitution directly via lookup tables becomes impossible; the LUT size and addressing mode is infeasible for the AVR ATtiny. A more viable alternative would be to first *extract* the bits required out of the bitsliced representation, *i.e.* temporarily revert to the original form (un-bitslicing), perform a lookup and then store back in the bitsliced representation. Still, this procedure also implies a large performance overhead.

The best solution that has been identified so far for computing efficiently the substitution layer of a cipher in bitsliced representation is by interpreting the S-box as a boolean function. Bitslicing uses 8-bit cells (Sect. 2.1), each pertaining to a position withing the cipher block. When implementing any boolean function under bitslicing, we still maintain the SIMD parallelization, *i.e.* any logical operation between two 8-way-bitsliced cells performs 8 single-bit logical operations in parallel.

Efficient software implementation of boolean functions. In order to efficiently implement a boolean function in software we point out its close resemblance to hardware construction of optimal circuits; in fact, we will demonstrate that boolean function implementation in software can be solved using the same techniques, albeit with slightly different constraints. Constructing optimal combinational circuits and 'technology mapping' in general is an intractable problem under almost any meaningful metric (gate count, depth, energy consumption, *etc.*). In practice, even a boolean function with as few as 8 inputs and a single output would require searching over a space of 2^{256} such outputs and this naturally leads us to heuristic methods.

Boyar-Peralta heuristic and Courtois extension. In 2008, Boyar and Peralta introduced an efficient new heuristic methodology to minimize the complexity of digital circuits [2,14,15]. Their focal point was to construct efficient cipher

[3] Instruction Set Architectures.

implementations based on the notion of *Multiplicative Complexity* (number of AND gates) and they produced a 2-stage methodology to optimize the circuit over the basis $\{\oplus, \wedge, 1\}$ by first minimizing the non-linear (AND) components and consequently the linear (XOR) components.

Courtois, Hulme and Mourouzis [19] extended this conjecture and applied the heuristic to several S-boxes modeled by $GF(2)^4 \rightarrow GF(2)^4$ boolean functions (including the PRESENT cipher S-box). In addition to the existing multiplicative complexity metric, Courtois *et al.* introduced the notion of *Bitslice Gate complexity* as the minimum number of 2-input gates of types XOR, OR, AND and single-input gates of type NOT needed to construct the circuit. For a silicon implementation this notion is helpful but definitely non-optimal: certain gates are more costly to implement, given the fact that silicon mapping often tries to minimize the number of the cheap NAND gates. Still, *we observe a case where software-efficient boolean functions differentiate from hardware-efficient boolean functions.* AVR ATtiny instructions for XOR, OR, AND, NOT operations cost a single clock cycle whereas there exists no native NAND operation. Consequently, mapping the PRESENT S-boxes to XOR, OR, AND, NOT gates and translating to software instructions outperforms any hardware-oriented mapping to NAND gates and subsequent translation to software operations. In the 'technology mapping' context, we can view these two approaches as mappings to different cell libraries, where the different component cost indicates the difference between hardware and software implementation.

The results of the Courtois form the basis of an efficient software-based bitsliced implementation of the PRESENT cipher, both for the AVR architecture (this work) and C-based implementations [35]. Courtois applied the 2-stage Boyar-Peralta heuristic in combination with SAT solvers, resulting in the following representation for the PRESENT Sbox that has very low bitsliced gate complexity.

```
T1=X2^X1; T2=X1&T1; T3=X0^T2; Y3=X3^T3; T2=T1&T3; T1^=Y3; T2^=X1;
T4=X3|T2; Y2=T1^T4; X3=~X3; T2^=X3; Y0=Y2^T2; T2|=T1; Y1=T3^T2;
```

where X_i=`input`, Y_i=`output` and T_i=`intermediate values`.

This is the final form that we use for computing the PRESENT substitution layer in the AVR ATtiny architecture and it requires 14 gates. Note that the set of operations uses the '`operator destination, sourceA, sourceB`' instruction format instead of the native ATtiny '`operator destination, source`' format. The inherent problem is that it is not possible to reuse a computed value, unless we store it temporarily elsewhere. With careful register usage, we maintain this penalty to a minimum and our final implementation requires 19 clock cycles to compute the output of a single PRESENT S-box. As a result, the 16 S-box operations used in the bitsliced representation require $19 \cdot 16 = 304$ clock cycles for 8 cipher blocks in parallel.

Table 3. Size (pertaining to flash and SRAM bytes) and throughput (clock cycles per block) of AES (row 1) and PRESENT (rows 2 to 5) cipher implementations.

Implementation	Flash (bytes)	SRAM (bytes)	Throughput (cc/block)	Bitsliced
AES, [1]	3098	-	2474	no
Eisenbarth *et al.* [21]	1000	18	10723	no
Papag. [38,39] ATtiny45	1794	0	8721	no
Rauzy *et al.*, ATtiny45 [41]	1194	144	6473	yes
This work, ATtiny85	3816	256	2967	yes

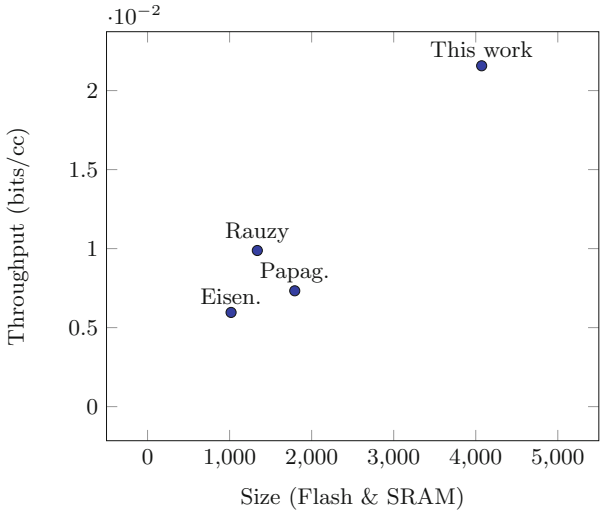

Fig. 2. Throughput *vs.* Size diagram for various implementations of the PRESENT cipher.

2.3 PRESENT Performance

The suggested implementation manages to outperform all existing implementations with respect to throughput. Comparing this work with the non-bitsliced work by Eisenbarth *et al.* [21], we can draw several conclusions regarding bitsliced representations. Eisenbarth's substitution layer is extremely efficient, consisting of a single flash memory lookup (4 clock cycles) per 8 bits (0.5 cc[4] per bit). Our boolean-function-oriented implementation requires 19 clock cycles for an S-box computation, *i.e.* 0.59 cc per bit, so slightly slower. However, this hindrance is unimportant when considering the very slow permutation layer of Eisenbarth *et al.* (154 cc per round) compared to ours (32 cc per round). We also outperform Papagiannopoulos and Verstegen [39] due to the fact that they replace the whole

[4] Clock cycles.

SP network with lookup tables and this results in a large number of flash memory accesses (1 memory access per 2 bits of state). However, we must stress the fact that the bitsliced version of PRESENT increased the memory requirements by a factor of 4, when compared to straightforward implementations [21]. Comparing our bitsliced version with Rauzy *et al.* [41], we observe that we achieve a *2.1*× boost in throughput. Since the authors do not elaborate on the implementation of the boolean function in use, memory accesses or other secondary operations (addRoundKey, keyUpdate *etc.*) we cannot identify the source of this speed-up, although we note that the authors were more efficient in terms of code size. When examining latency, we note that all bitsliced implementations perform inherently multiple blocks in parallel (equal to the bitslice factor). In our case, we perform 8 block encryptions in parallel within 23736 clock cycles, resulting in poor latency performance. It is also worth pointing out that AES [1] can outperform PRESENT in terms of both latency and throughput, since it encrypts a 128-bit block (twice the PRESENT block) in fewer cycles (Fig. 2 and Table 3).

3 PRINCE: Nibble-Slicing in 8-Bit Microprocessors

In this section, we present the first (to our knowledge) 'sliced' implementation of the PRINCE cipher [13] for the ATtiny architecture. Our focal points are the substitution and the nibble (4 bits) permutation operations of the cipher. We suggest a novel idea, namely a variation of the bitslicing technique called nibble-slicing, in order to efficiently compute these operations. We also offer insight w.r.t. the effects of slicing on the permutation and substitution layer and provide a comparison between bitslicing (used in PRESENT) and nibble-slicing (used in PRINCE).

Algorithm outline. PRINCE is a 64-bit block cipher with a 128-bit keys, based on the F–X construction [13,31]. The key k is split into two parts of 64 bits each, *i.e.* $k = k_1||k_2$ and extended to 192 bits via the following mapping:

$$k_0||k_1 \rightarrow k_0||k_0'||k_1 = (k_0||k_0 >>> 1) \oplus (k_0 >> 63||k_1) \tag{1}$$

Now, k_0 and k_0' are used as whitening keys, while k_1 is the main 64-bit used by the 12 rounds of the cipher without any key updates. Figure 3 shows the 12 rounds of encryption. The encryption consists of a nibble-based substitution layer S, a Shift Rows operation (SR) and a matrix multiplication M'. Operations M' and SR (in this order) construct the operation M (Tables 4, 5).

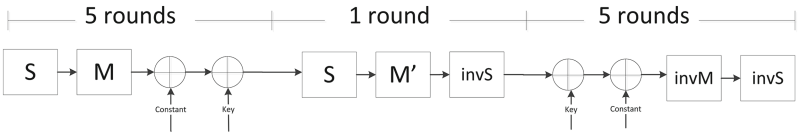

Fig. 3. The 12 rounds of the PRINCE cipher. k_1 denotes the core cipher key, RC_is are constants, S the substitution layer and M the diffusion layer.

Table 4. The S-Box of the PRINCE cipher, used for the S operation.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S[x]	B	F	3	2	A	C	9	1	6	7	8	0	E	5	D	4

Table 5. Nibble permutation of the PRINCE cipher in the SR operation (from old nibble position to new nibble position).

Old	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
New	0	5	10	15	4	9	14	3	8	13	2	7	12	1	6	11

3.1 Diffusion Layer Under Nibble-Slicing

Shift Rows. When comparing the substitution-permutation network of PRESENT with that of PRINCE we can observe similarities and differences. The substitution operation is fairly identical in nature and similarities do exist between the Shift Rows operation (nibble permutation) and the PRESENT bit permutation network; the SR operation is a permutation with fewer degrees of freedom when compared to single-bit permutations. Based on this observation, we have identified a technique stemming from bitslicing (we call it *nibble-slicing*) that is custom-made for nibble-oriented permutation layers and manages to avoid memory accesses, despite the fact that we operate on a 64-bit cipher state (Fig. 4).

$$M' = \begin{pmatrix} M_a & 0 & 0 & 0 \\ 0 & M_b & 0 & 0 \\ 0 & 0 & M_b & 0 \\ 0 & 0 & 0 & M_a \end{pmatrix}$$

$$M_a = \begin{pmatrix} M_0 & M_1 & M_2 & M_3 \\ M_1 & M_2 & M_3 & M_0 \\ M_2 & M_3 & M_0 & M_1 \\ M_3 & M_0 & M_1 & M_2 \end{pmatrix} \quad M_b = \begin{pmatrix} M_1 & M_2 & M_3 & M_0 \\ M_2 & M_3 & M_0 & M_1 \\ M_3 & M_0 & M_1 & M_2 \\ M_0 & M_1 & M_2 & M_3 \end{pmatrix}$$

$$M_0 = \begin{pmatrix} 0&0&0&0 \\ 0&1&0&0 \\ 0&0&1&0 \\ 0&0&0&1 \end{pmatrix} \quad M_1 = \begin{pmatrix} 1&0&0&0 \\ 0&0&0&0 \\ 0&0&1&0 \\ 0&0&0&1 \end{pmatrix} \quad M_2 = \begin{pmatrix} 1&0&0&0 \\ 0&1&0&0 \\ 0&0&0&0 \\ 0&0&0&1 \end{pmatrix} \quad M_3 = \begin{pmatrix} 1&0&0&0 \\ 0&1&0&0 \\ 0&0&1&0 \\ 0&0&0&0 \end{pmatrix}$$

Fig. 4. The M' operation, analyzed from top to bottom.

Nibble-slicing uses the following representation: every 8-bit register is split into two parts (high and low, 4 bits each) and we use a total of 16 registers (thus avoiding SRAM usage, something impossible for bitsliced representations on ATtiny). The whole representation consists of 128 bits, *i.e.* two separate cipher states (we refer to them as block 1 and block 2 – see Fig. 5). Block 1 is stored in all high parts of the 16 registers and block 2 in all low parts of the corresponding registers. Nibble-slicing presents similarities with vectorized computations on

larger processors and to digit-slicing or byte-slicing techniques used to improve speed of AES [30]. In our context, nibble-slicing essentially removes the need to compute the SR operation and could be of similar usage for other lightweight ciphers with a nibble-oriented permutation network (*e.g.* KLEIN or LED).

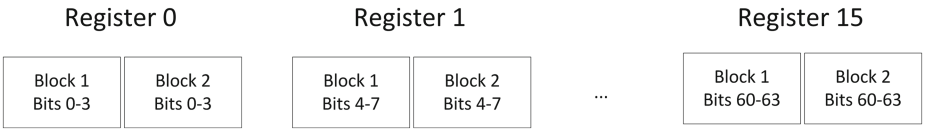

Fig. 5. Nibble-sliced representation of two PRINCE cipher blocks.

Matrix multiplication. Matrix multiplication (M') is the most computationally expensive operation of the PRINCE cipher. To increase speed, we try to exploit the diagonal structure of the matrix: we view the matrix as a set of 4 by 4 matrices, then we multiply with the state nibbles with the main diagonal of every 4 by 4 matrix. This approach works well under the nibble-sliced representation; both high and low parts of the register are multiplied with the same diagonal.

3.2 Substitution Layer Under Nibble-Slicing

A negative effect of nibble-slicing is the following: under this non-standard representation, we have lost our maximum parallel processing capability; instead of storing 8 different cipher states within a single register (bitslice factor of 8) we store only two (bitslice factor of 2). However, this novel representation is faster when implementing PRINCE in the AVR context compared to the original bitslicing method for the following reasons:

1. As mentioned, nibble-slicing in 16 registers results in an implementation that fully avoids usage of SRAM and the penalty associated with it. Storing two separate cipher states in such a way fits into registers and thus avoids spills to SRAM.
2. Second, although it is still possible, we no longer have to compute the S-box via a boolean function and we can use LUTs which are more efficient in the ATtiny context.

Although we demonstrated in Sect. 2.2 that boolean functions are fairly efficient for S-box computation, we remind that they are still slower than direct flash memory lookups. Bitsliced PRESENT could not use lookup tables for the substitution layer, but that is not the case for nibble-sliced PRINCE. Each register contains two separate 4-bit values. Based on the guidelines by Eisenbarth *et al.* [21] and Papagiannopoulos and Verstegen [39], we use a 'squared', byte-oriented lookup table for S-box computation. During the lookup, each one of

Table 6. Performance of the high-throughput of the AES (row 1) and PRINCE (rows 2 to 4) ciphers.

Implementation	Flash (bytes)	SRAM (bytes)	Throughput (cc/block)
AES [1]	3098	-	2474
Shahverdi *et al.*, T-box	1990	232	4292
Shahverdi *et al.*, parallel	1574	24	3253
This work, ATtiny85	2382	220	1803

the 4-bit halves is substituted separately. The whole process is carried out efficiently via 8-bit flash memory lookups from 256-byte tables in flash memory. In fact, we merge the S operation with the SR operation; every time we perform a lookup, we take into account that values need to be stored back in registers in a permuted fashion.

3.3 PRINCE Performance

Nibble-slicing lacks in terms of throughput compared to the 'traditional' bitslicing approach. However, the fact that LUTs are a viable option for the substitution layer compensates to some extent. Our PRINCE cipher implementation encrypts two 64-bit blocks in 3606 cc, *i.e.* a throughput of 1803 cc per block. Comparing to a straightforward implementation that uses T-tables (Shahverdi *et al.* [5]), we observe a throughput increase of 2.3, while memory consumption increased by a factor of 1.16. Comparing to a parallel PRINCE implementation (Shahverdi *et al.* [5]), we achieve throughput increase by a factor of 1.8 and memory requirements increase by a factor of 1.61. AES [1] still outperforms PRINCE (0.051 bits/cc *vs.* 0.035 bits/cc) (Table 6).

4 KATAN64: Hardware Parallelism Translated to Software Slices

The third section of this work examines a different type of cipher that is not related to SP networks and resembles a stream cipher. However, as we will point out, certain parallel constructs in hardware can also lead us to a non-standard representation in software that taps into parallelism – not unlike bitslicing. We identify these cases as 'hardware slices'.

Algorithm outline. The outline is provided in Fig. 6. The KATAN cipher [16] was designed as a secure 80-bit block cipher with a minimal number of hardware gates, while it demonstrates very slow software performance. Following the design of KeeLoq [22], the designers chose a structure similar to a stream cipher, resembling the two-register variant of Trivium [17], known as Bivium.

The cipher's plaintext is loaded into two linear feedback shift registers (LFSRs) L1 and L2. Each round several bits are taken from the registers and the cipher

key. Those bits enter two non-linear boolean functions (f_a and f_b), while the output of the boolean functions is loaded to the least-significant bits of the registers after they are shifted (or 'clocked'). Computing the two boolean functions f_a, f_b requires AND and XOR operations between the state bits, the cipher keys and a constant value IR (irregular update) that increases diffusion. The KATAN cipher executes a fairly large number of rounds (254) and comes in three variants: KATAN32, KATAN48 and KATAN64 (the suffix denotes the size of the cipher state – the key size is always 80 bits). Our implementation focuses solely on the 64-bit version, which presents additional interest w.r.t. slicing.

As mentioned, KATAN64 uses two non-linear function f_a and f_b in each round which are computed as follows.

$$f_a(L_1) = L_1[24] \oplus L_1[15] \oplus (L_1[20] \cdot L_1[11]) \oplus (L_1[9] \cdot IR) \oplus k_a \quad (2)$$

$$f_b(L_2) = L_2[38] \oplus L_2[25] \oplus (L_2[33] \cdot L_2[21]) \oplus (L_2[14] \cdot L_2[9]) \oplus k_b \quad (3)$$

where $L_1[i]$ and $L_2[i]$ denote bit positions on the two LFSR registers, IR denotes the irregular update (constant) and k_a, k_b denote the two subkey bits of every KATAN64 round. After the computation of the non-linear functions, the registers L1 and L2 are shifted. The MSB falls off into the corresponding non-linear function and the LSB is loaded with the output of the second non-linear function, *i.e.*, after the round, the LSB of L1 is the output of f_b and the LSB of L2 is the output of f_a.

A specific feature of the KATAN64 construction with respect to the non-linear functions is the following. In KATAN64, each round applies f_a and f_b *three times* with the same key bits k_a, k_b. An efficient hardware implementation can implement these three steps in parallel, a fact that will also lead us to software parallelism.

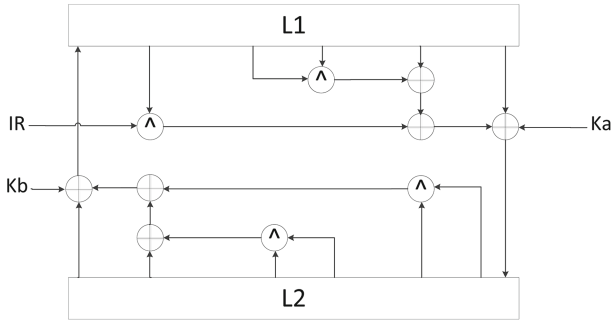

Fig. 6. The core operation of the KATAN cipher. The two LFSR L1, L2 store the cipher state. Several bits are extracted from L1, L2, from the cipher key (k_a, k_b) and from IR in order to compute the non-linear functions f_a, f_b (via XOR/AND operations) and to update the cipher state.

The key schedule of the KATAN64 cipher loads the 80-bit key into an LFSR (the least significant bit of the key is loaded to position 0 of the LFSR). Every round, positions 0 and 1 of the LFSR are used as the round's subkey k_{2i} and k_{2i+1}, and the LFSR is clocked twice according to the following feedback polynomial:

$$x^{80} + x^{61} + x^{50} + x^{13} + 1 \tag{4}$$

The subkey of round i can be described as $k_a||k_b = k_{2i}||k_{2i+1}$ where $k_i = K_i$ for $i \in \{0, 1, \ldots, 79\}$ (K being the 80-bit input key) or alternatively $k_i = k_{i-80} \oplus k_{i-61} \oplus k_{i-50} \oplus k_{i-13}$.

4.1 KATAN64 Non-linear Functions Under Slicing

The KATAN cipher has an interesting hardware-related property that has not been yet translated to software implementations. During each cipher round, the 64-bit version of KATAN applies the non-linear functions f_a, f_b three times and these computations can be carried out in parallel (if the extra hardware gates are available). Eisenbarth *et al.* suggest that implementing this property may result in complicated shifting/masking that will increase the code size with little or no performance gain, yet we attempt to rebut this statement.

Computing the functions f_a, f_b sequentially via the `bld,bst` bit-level instructions is very time-consuming. A single run of f_b would require 7 extract (`bld`), 7 deposit (`blst`), 2 AND, 3 XOR operations and as a result $3 \cdot 19 \cdot 254 = 14478$ clock cycles for a full encryption (the factor 3 due to the 3-way parallelizable step being done sequentially). Analogously, f_a also costs roughly the same amount. Bitslicing would solve this issue but it would entail a huge SRAM transfer overhead due to the large number of rounds. Thus, we turn to register-oriented approaches.

Achieving 3-way parallelizability involves using masking and instructions that operate on register level and not bit-level operations. In addition, it involves a slightly different representation of the cipher state: instead of storing the 64 bit state in 8 registers (each containing 8 bits), we employ 9 registers that store the representation in a slid fashion (see Fig. 7). First, observe that there exist several triadic bit groups that contribute to the computation of the next cipher state. For instance, KATAN64 uses (among others) bit 9 of the the L2 LFSR to compute a single bit of the next state and since this operation has to be carried out 3 times within a KATAN64 round, the same procedure is applied to bits 8 and 7 correspondingly. There exist 6 such triads in the L2 LFSR (9/8/7, 14/13/12, 21/20/19, 25/24/23, 33/32/31, 38/37/36) and 5 such triads in the L1 LFSR (9/8/7, 11/10/9, 15/14/13, 20/19/18, 24/23/22). This non-standard representation displayed in Fig. 7 attempts to arrange all bit triads used for the new state computation in a way that never splits a triad between two separate registers. Having established that, we can use register-level operations that carry out the new state computations, while maintaining 3-way parallelizability. We have essentially created 3 slices in our representation.

Under the new representation, computing *3 parallel output bits* costs 19 clock cycles for function f_b and 19 clock cycles for function f_a. Compared to the

sequential approach of the previous paragraph, we observe a $3\times$ performance boost when parallelizing the operations in software; f_a and f_b used to cost $57 = 3{\cdot}19$ cycles each for a 3 bit output. Note also that the new representation does not fully utilize all registers, since registers `r0`, `r5` and `r8` have bits indicated as *null* (*i.e.* non-relevant in our representation). A side-effect is that bit rotation (also denoted as LFSR clocking) becomes slightly slower; it costs us 39 clock cycles in order to carry out 3 bit rotations to all 9 registers that are transparent to the *null* register positions, *i.e.* sliding all registers to the right and transferring overflow bits from L2 to L1 and L1 to L2 correspondingly while taking into account the null bits. A standard representation (using 8 registers without *null* bit elements) would rotate in 24 clock cycles ($24 = 3 \cdot 8$, *i.e.* 3 single bit rotations carried on 8 registers) Fig. 8.

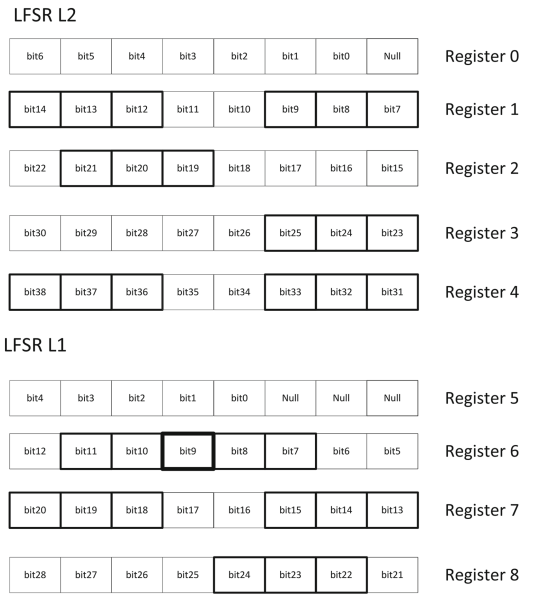

Fig. 7. Cipher state of KATAN64, stored in a slid manner, using 9 registers. The bit triads required for computing the new cipher state are highlighted in bold.

4.2 KATAN64 Performance

The only known implementation of KATAN64 in AVR architecture is presented by Eisenbarth *et al.* [21] and it focuses on low size, not high throughput. Our implementation manages a full KATAN64 encryption in 23671 clock cycles, while Eisenbarth *et al.* manages a full encryption in 72063 clock cycles, *i.e.* we improve the throughput by a factor of 3. Although the two implementations are not directly comparable (due to different implementation objectives) it is still useful to compare and observe the tradeoffs. Specifically, we disagree with the statement that the 3-way KATAN64 parallelizability cannot be sufficiently exploited

```
function fb          function fa

mov t1,s1             mov t3,s6
swap t1               lsr t3
lsr t1                eor t3,s8
and t1,s1

mov t2,s2             eor t3,s8
swap t2               lsr t3
and t2,s4             eor t3,s7

eor t1,t2             mov t4,s7
eor t1,s3             lsr t4
                      and t4,s6
swap t1               swap t4
lsl t1                eor t4,t3
eor t1, s4
```

Fig. 8. Register-oriented code to compute f_a, f_b, while performing operations in parallel (excluding key XOR operations and irregular update XORing). Variables s_i denote cipher state (Fig. 7 register i corresponds to s_i), and variable t_j denotes temporary values.

in software; with the penalty of a single extra register, we manage to increase the throughput of the non-linear layer threefold. Although we exploit a form of parallelizability, we do not compute many blocks in parallel; thus, throughput improvement translates automatically to latency improvement. Finally, our implementation precomputes the cipher round key and requires extra SRAM space for lowering the latency (Table 7).

Table 7. Throughput of KATAN64 cipher implementations for AVR architecture, *i.e.*, clock cycles required for a single encryption round.

KATAN64 implementation	Throughput (cc)	Size (bytes)
Eisenbarth *et al.* [21]	72063	338 flash, 18 SRAM
This work, ATtiny45	23671	380 flash, 96 SRAM

5 Conclusion

Summarizing, this work has managed to improve the throughput aspect of three lightweight ciphers (PRESENT, PRINCE, KATAN64). We displayed the 'slicing' techniques, then determined which is applicable for each cipher and finally, we investigated their effects on substitution, permutation and other operations.

Our results demonstrate a 2.1× improvement for PRESENT throughput, 3× improvement for KATAN64 (throughput and latency) and the first high-speed implementation of PRINCE for ATtiny devices. Code is available online here: https://github.com/kostaspap88?tab=repositories. Future directions include high-throughput implementations for ciphers or hash functions that present structural similarities with the three ciphers discussed in this paper. Finally, an interesting direction would be an attempt to analytically model the behavior of *e.g.* SP networks and rigidly link computational efficiency in software with other important properties such as cryptanalysis resistance, power consumption and hardware performance. Establishing trade-offs between these design parameters in a analytic manner could link to more efficient designs in the future.

Acknowledgments. We would like to thank P. Schwabe and the rest of the reviewers of this paper for their contribution. This work was supported in part by the Technology Foundation STW (project 12624 – SIDES), The Netherlands Organization for Scientific Research NWO (project ProFIL – 628.001.007) and the ICT COST action IC1204 TRUDEVICE.

References

1. Avr, AES: The AES block cipher on AVR controllers. http://point-at-infinity.org/avraes/
2. Circuit minimization results obtained at Yale University. http://cs-www.cs.yale.edu/homes/peralta/CircuitStuff/CMT.html. (Accessed 15 November 2013)
3. ISO/IEC 29192–2:2011, Information technology - Security techniques - Lightweight cryptography - Part 2: Block ciphers (2011)
4. Abed, F., Forler, C., List, E., Lucks, S., Wenzel, J.: Biclique cryptanalysis of the PRESENT, LED and KLEIN. IACR Cryptol. ePrint Arch., 591 (2012). http://eprint.iacr.org/2012/591.pdf. (Accessed 18 November 2013)
5. Aria, S., Eisenbarth, T.: AVRprince - An Efficient Implementation of PRINCE for 8-bit Microprocessors. http://users.wpi.edu/teisenbarth/pdf/avrPRINCEv01.pdf.
6. Aumasson, J-P., Henzen, L., Meier, W., Naya-Plasencia, M.: Quark: A lightweight hash. J. Cryptol. 26(2), 313–339 (2013). https://131002.net/quark/quark_full.pdf. (Accessed 18 November 2013)
7. Babbage, S., Dodd, M.: The stream cipher MICKEY 2.0, ECRYPT stream cipher (2006). http://www.ecrypt.eu.org/stream/p3ciphers/mickey/mickey_p3.pdf. (Accessed 18 November 2013)
8. Biham, E.: A fast new DES implementation in software. In: Biham, E. (ed.) FSE 1997. LNCS, vol. 1267, pp. 260–272. Springer, Heidelberg (1997). http://citeseerx.ist.psu.edu/viewdoc/download?
9. Bishop, M.: An application of a fast data encryption standard implementation. Comput. Syst. 1(3): 221–254 (1988). http://www.cs.dartmouth.edu/reports/TR88-138.pdf. (Accessed 18 November 2013)
10. Bogdanov, A., Knežević, M., Leander, G., Toz, D., Varıcı, K., Verbauwhede, I.: SPONGENT: a lightweight hash function. In: Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917, pp. 312–325. Springer, Heidelberg (2011). http://homes.esat.kuleuven.be/abogdano/papers/spongent_ches11.pdf

11. Bogdanov, A.A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M., Seurin, Y., Vikkelsoe, C.: PRESENT: An Ultra-Lightweight Block Cipher. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 450–466. Springer, Heidelberg (2007). http://homes.esat.kuleuven.be/abogdano/papers/present_ches07.pdf
12. Bogdanov, A., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J.B., Seurin, Y.: Hash functions and RFID tags: mind the gap. In: Oswald, E., Rohatgi, P. (eds.) CHES 2008. LNCS, vol. 5154, pp. 283–299. Springer, Heidelberg (2008). http://www.iacr.org/archive/ches2008/51540279/51540279.pdf
13. Borghoff, J., Canteaut, A., Güneysu, T., Kavun, E.B., Knezevic, M., Knudsen, L.R., Leander, G., Nikov, V., Paar, C., Rechberger, C., Rombouts, P., Thomsen, S.S., Yalçın, T.: PRINCE – a low-latency block cipher for pervasive computing applications. In: Wang, X., Sako, K. (eds.) ASIACRYPT 2012. LNCS, vol. 7658, pp. 208–225. Springer, Heidelberg (2012). http://eprint.iacr.org/2012/529.pdf
14. Boyar, J., Peralta, R.: A new combinational logic minimization technique with applications to cryptology. In: Festa, P. (ed.) SEA 2010. LNCS, vol. 6049, pp. 178–189. Springer, Heidelberg (2010). http://eprint.iacr.org/2009/191.pdf
15. Boyar, J., Peralta, R.: A small depth-16 circuit for the AES S-box. In: Gritzalis, D., Furnell, S., Theoharidou, M. (eds.) SEC 2012. IFIP AICT, vol. 376, pp. 287–298. Springer, Heidelberg (2012). http://eprint.iacr.org/2011/332.pdf
16. De Cannière, C., Dunkelman, O., Knežević, M.: KATAN and KTANTAN — a family of small and efficient hardware-oriented block ciphers. In: Clavier, C., Gaj, K. (eds.) CHES 2009. LNCS, vol. 5747, pp. 272–288. Springer, Heidelberg (2009). http://www.cs.technion.ac.il/orrd/KATAN/CHES2009.pdf
17. De Cannière, C., Preneel, B.: Trivium. In: Robshaw, M., Billet, O. (eds.) New Stream Cipher Designs. LNCS, vol. 4986, pp. 244–266. Springer, Heidelberg (2008). http://www.ecrypt.eu.org/stream/p3ciphers/trivium/trivium_p3.pdf
18. Collard, B., Standaert, F.-X.: A statistical saturation attack against the block cipher PRESENT. In: Fischlin, M. (ed.) CT-RSA 2009. LNCS, vol. 5473, pp. 195–210. Springer, Heidelberg (2009)
19. Courtois, N., Hulme, D., Mourouzis, T.: Solving circuit optimisation problems in cryptography and cryptanalysis. IACR Cryptol. ePrint Arch. 2011:475 (2011). http://www.ima.org.uk/_db/_documents/Courtois.pdf. (Accessed 18 November 2013)
20. Atmel datasheet. Atmel 8-bit AVR microcontroller datasheet. http://tinyurl.com/klld65e. (Accessed 18 November 2013)
21. Eisenbarth, T., Gong, Z., Güneysu, T., Heyse, S., Indesteege, S., Kerckhof, S., Koeune, F., Nad, T., Plos, T., Regazzoni, F., Standaert, F.-X., van Oldeneel tot Oldenzeel, L.: Compact implementation and performance evaluation of block ciphers in attiny devices. In: Mitrokotsa, A., Vaudenay, S. (eds.) AFRICACRYPT 2012. LNCS, vol. 7374, pp. 172–187. Springer, Heidelberg (2012). http://perso.uclouvain.be/fstandae/PUBLIS/108.pdf
22. Eisenbarth, T., Kasper, T., Paar, C., Indesteege, S.: Encyclopedia of Cryptography and Security, 2nd edn. Springer (2011)
23. Flynn, M.J.: Some computer organizations and their effectiveness. IEEE Trans. Comput. C-21(9): 948–960, September (1972)
24. Gomathisankaran, M., Lee, R.B.: Maya: a novel block encryption function (2009). http://palms.princeton.edu/system/files/maya.pdf. (Accessed 18 November 2013)
25. Gong, Z., Nikova, S., Law, Y.W.: KLEIN: a new family of lightweight block ciphers. In: Juels, A., Paar, C. (eds.) RFIDSec 2011. LNCS, vol. 7055, pp. 1–18. Springer, Heidelberg (2012). http://doc.utwente.nl/73129/1/The_KLEIN_Block_Cipher.pdf

26. Guo, J., Peyrin, T., Poschmann, A.: The PHOTON family of lightweight. In: Rogaway, p (ed.) Advances in Cryptology – CRYPTO 2011. LNCS, vol. 6841, pp. 222–239. Springer, Heidelberg (2011). http://www.ecrypt.eu.org/hash2011/proceedings/hash2011_04.pdf
27. Guo, J., Peyrin, T., Poschmann, A., Robshaw, M.: The LED block cipher. In: Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917, pp. 326–341. Springer, Heidelberg (2011). http://eprint.iacr.org/2012/600.pdf
28. Hell, M., Johansson, T., Meier, W.: Grain: a stream cipher for constrained environments. Int. J. Wireless Mobile Comput. 2, 86–93 (2007). http://www.ecrypt.eu.org/stream/ciphers/grain/grain.pdf. (Accessed 18 November 2013)
29. Hong, D., Sung, J., Hong, S.H., Lim, J.-I., Lee, S.-J., Koo, B.-S., Lee, C.-H., Chang, D., Lee, J., Jeong, K., Kim, H., Kim, J.-S., Chee, S.: HIGHT: a new block cipher suitable for low-resource device. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 46–59. Springer, Heidelberg (2006). http://www.iacr.org/cryptodb/archive/2006/CHES/04/04.pdf
30. Käsper, E., Schwabe, P.: Faster and timing-attack resistant AES-GCM. In: Clavier, C., Gaj, K. (eds.) CHES 2009. LNCS, vol. 5747, pp. 1–17. Springer, Heidelberg (2009). http://eprint.iacr.org/2009/129
31. Kilian, J., Rogaway, P.: How to protect DES against exhaustive eey search. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 252–267. Springer, Heidelberg (1996). http://www.cs.ucdavis.edu/rogaway/papers/desx.pdf
32. Könighofer, R.: A fast and cache-timing resistant implementation of the AES. In: Malkin, T. (ed.) CT-RSA 2008. LNCS, vol. 4964, pp. 187–202. Springer, Heidelberg (2008). https://online.tugraz.at/tug_online/voe_main2.getvolltext?pCurrPk=47852
33. Leander, G.: Small scale variants of the block cipher present. IACR Cryptol. ePrint Arch. 2010:143 (2010). http://eprint.iacr.org/2010/143.pdf. (Accessed 18 November 2013)
34. Lee, R.B., Shi, Z., Yang, X.: Cryptography efficient permutation instructions for fast software. IEEE Micro. 21, 56–69 (2001). http://palms.ee.princeton.edu/PALMSopen/lee01efficient.pdf. (Accessed 18 November 2013)
35. Hulme, D., Song, G., Albrecht, M., Courtois, N.T.: Bit-slice implementation of PRESENT in pure standard C. https://bitbucket.org/malb/research-snippets/src. (Accessed 18 November 2013)
36. Nakahara Jr., J., Sepehrdad, P., Zhang, B., Wang, M.: Linear (hull) and algebraic cryptanalysis of the block cipher PRESENT. In: Garay, J.A., Miyaji, A., Otsuka, A. (eds.) CANS 2009. LNCS, vol. 5888, pp. 58–75. Springer, Heidelberg (2009). http://www.ioc.ee/tarmo/tday-meintack/zhang-slides.pdf
37. Papagiannopoulos, k.: Present with bld, bst instructions. https://github.com/kostaspap88/sc_res_present. (Accessed 18 November 2013)
38. Papagiannopoulos, K.: Speed-optimized implementation of PRESENT in AVR assembly (2013). https://github.com/kostaspap88/PRESENT_speed_implementation/ (Accessed 18 November 2013)
39. Papagiannopoulos, K., Verstegen, A.: Speed and size-optimized implementations of the PRESENT cipher for tiny AVR devices. In: Hutter, M., Schmidt, J.-M. (eds.) RFIDsec 2013. LNCS, vol. 8262, pp. 159–173. Springer, Heidelberg (2013)
40. Pornin, T.: Automatic software optimization of block ciphers using bitslicing techniques.http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.48.3085&rep=rep1&type=pdf.

41. Rauzy, P., Guilley, S., Najm, Z.: Formally proved security of assembly code against leakage. IACR Cryptol. ePrint Arch. 554 (2013). http://eprint.iacr.org/2013/554.pdf. (Accessed 18 November 2013)
42. Shi, Z., Lee, R.B.: Bit permutation instructions for accelerating software cryptography. In: Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors, pp. 138–148, 2000. http://www.princeton.edu/rblee/PUpapers/shi_asap00.pdf
43. Shirai, T., Shibutani, K., Akishita, T., Moriai, S., Iwata, T.: The 128-bit block-cipher CLEFIA (extended abstract). In: Biryukov, A. (ed.) FSE 2007. LNCS, vol. 4593, pp. 181–195. Springer, Heidelberg (2007). http://www.iacr.org/archive/fse2007/45930182/45930182.pdf