Radboud University Nijmegen

Kerckhoffs Institute

Master Thesis

High-throughput implementations of lightweight ciphers in the AVR ATtiny architecture.

Author:
Kostas
Papagiannopoulos

Supervisors:
Dr. Peter Schwabe
Dr. Lejla Batina

November 18, 2013

Contents

1 Introduction

During the recent years, our society experienced large 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. Mundane, 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, 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 (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 [41]), has led researchers to establish new ciphers that are tailor-made for pervasive computing and are often referred to as lightweight ciphers.

1.1 Lightweight ciphers

Lightweight cryptographic primitives add a new dimension in cryptographic primitive construction; hardware cost becomes now an important design element. Ciphers still need to address cryptographic security (attack resilience) but they must not ignore the hardware implementation cost.

These ciphers have drawn considerable attention during the recent years. Among the best studied algorithms are the block ciphers CLEFIA [57], Hight [37], KATAN, KTAN-TAN [15], Klein [30], mCrypton [46], LED [33],

Piccolo [56], PRESENT [10], NEOKEON citeneokeon, the stream ciphers Grain [34], Mickey [6], and Trivium [16] and more recently lightweight hash functions such as SPONGENT [9], PHOTON [32] and QUARK [5].
It is of particular interest to point out that the above-mentioned primitives present several similarities and differences in the way their primary components are constructed. For instance, KATAN designers attempted a minimalistic design (inspired by stream ciphers) resulting in hardware implementations that are extremely low in gate count, yet not particularly fast. The PRESENT designers opted for an Substitution-Permutation (SP) structure that can achieve fairly low gate count but also scalability with respect to hardware performance and they provide us with a gate number *vs.* speed tradeoff. More recently, a new class of ciphers emerged, that of high-speed hardware ciphers like PRINCE [12], which attempts to include the low execution latency parameter into cipher construction.
In order to address those three different hardware-related goals (security, size, speed, see Figure 1), researchers proposed several constructs with different capabilities that either combine features or attempt to maximize specific goals.

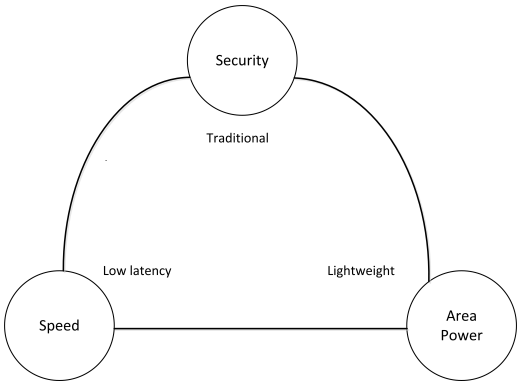

Figure 1: The three goals that affect the design of cryptographic primitives. High security relates to traditional cryptography, low area/power to lightweight cryptography and high speed to low-latency goals.

To complicate matters even further, we point out that all those crypto-enabled hardware devices often need to communicate with components that run software and thus, we can introduce the software implementation dimension in our discussion. Unfortunately, hardware-related properties such as the speed or size of a cipher developed with hardware implementations in

mind do not automatically or conclusively translate to software implementation features. In fact, we have observed some hardware-oriented ciphers to be substantially slower in software than AES. Design choices that achieve hardware performance, like 4-bit-based diffusion matrix components in PRINCE do not inherently imply software performance. For instance, using 8-bit-based diffusion matrix components in KLEIN can achieve better software performance due to the instruction capabilities of 8-bit microcontrollers. In addition, the extent to which we can construct fast lookup tables for the cipher is directly linked to software performance, despite the fact that it is not very important when implementing ASICs[1].

Our contribution

This work examines thoroughly two lightweight, hardware-oriented block ciphers (PRESENT and KATAN) from a software perspective (chapters 2, 3). Specifically, we construct software implementations of these two ciphers on AVR ATtiny microcontrollers that achieve higher throughput than the current state of the art. In addition, under the same perspective, we examine the structure of the fairly new PRINCE cipher (chapter 4) and establish a fast implementation of the core SP PRINCE network (excluding matrix-based diffusion from our analysis). We conclude in chapter 5.

1.2 AVR ATtiny architecture

A very common target for lightweight cipher implementations is the AVR architecture due to its small cost, reasonable performance and high versatility. Thus, in the current section, we provide the reader with an overview of the AVR architecture and its instruction set, analyzing specific focal points that are important in our work scope.
The AVR is a modified Harvard architecture[2] 8-bit RISC[3] single chip microcontroller which was first developed by Atmel in 1996. There exist several classes of AVR microcontrollers, namely ATtiny, ATmega, ATXmega, AVR with FPGA and more recently, even 32-bit oriented AVR devices. Our implementations target the AVR ATtiny class, due to its particularly low cost which can assist and enable secure ubiquitous computing in a large range of applications via efficient implementation of lightweight ciphers.

[1] Application-specific integrated circuits.

[2] A computer architecture with physically separate storage and signal pathways for instructions and data.

[3] Reduced instruction set computing

The AVR ATtiny architecture [22, 21] contains several components, most importantly:

- ALU, the arithmetic-logic unit.
- Flash memory storage.
- SRAM, the static random-access memory.
- Input/output memory
- EEPROM, the electrically erasable programmable read-only memory. Due to its low speed it is not of implementation interest in this work.
- 32 internal general-purpose registers.

A schematic giving an overview of the ATtiny basic blocks is given in the Figure 2.

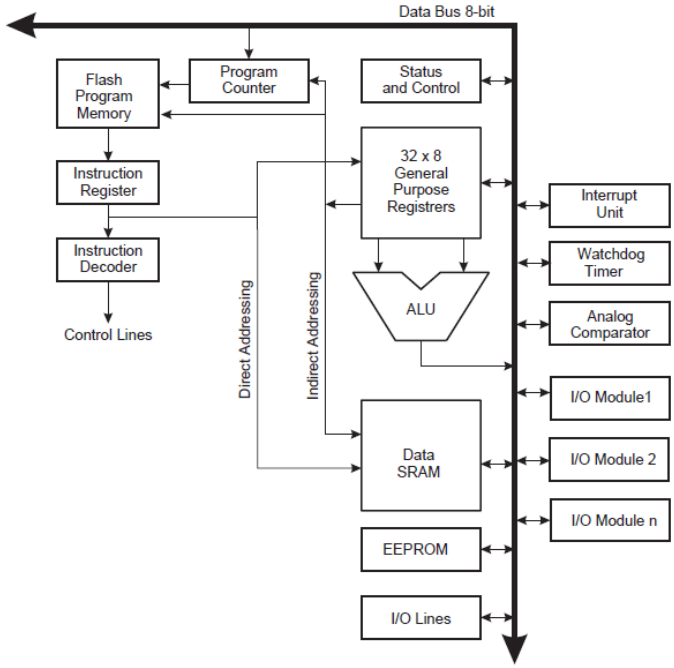

Figure 2: The main components of an AVR ATtiny microcontroller.

ATtiny instruction fetch

As mentioned, in order to maximize performance and parallelism, the AVR uses a Harvard architecture with separate memories and buses for program and data. Instructions in the program memory (reprogrammable flash memory) are executed with a two-stage pipelining, *i.e.*, while one instruction is being executed, the next instruction is pre-fetched from the program memory. This concept enables instructions to be executed in every clock cycle, since normally fetching from flash memory is much slower.

ATtiny ALU

The ATtiny ALU register file contains 32 8-bit general-purpose registers (`r0...r31`) with a single-clock-cycle access time. This allows single-cycle arithmetic logic unit operation with two operands, *i.e.* instructions have this form: `operator registerA, registerB`.Computation-wise, two operands are output from the register file, the operation is executed, and the result is stored back in the register file, all within one cycle. Most common logic and arithmetic operations can be performed in the AVR ATtiny architecture (including operations between a register and a constant) with the notable exception of the multiplication operation.
Six of the 32 registers (`r26...r31`) can be used as three 16-bit indirect address register pointers for data-space addressing enabling efficient address calculations that pertain to SRAM access. These added function registers are the 16-bit `X` (`r26,r27`), `Y` (`r28,r29`), and `Z` (`r30,r31`), where only the `Z` register pertains to flash memory access.
A flags register is also included and contains information about the result of the most recently executed arithmetic instruction (such information is usually used for altering program flow in order to perform conditional operations). Of particular interest to this work is the status register flag T, used for bit-copy storage. The bit copy instruction `bld` and bit storage instruction `bst` use the T-bit as source or destination for the operand bit. A bit from a register in the register file can be copied into T by the `bld` instruction, and a bit in T can be copied into a bit in a register in the register file by the `bst` instruction, effectively generating bit-extract and bit-deposit operations that can permute bits.
Moreover, ATtiny supports IO with an global-interrupt-enable bit in the status register for flexibility and also provides us with a stack memory space that uses SRAM and enables subroutine calls.

AVR memories

The ATtiny25/45/85 microporcessor models contain 2/4/8KB of on-chip reprogrammable flash memory for program storage. Since all AVR instructions are 16 or 32 bits wide, the flash memory is organized as 1024/2048/4096 $\times$ 16 bits. It is of special interest to point out that constant tables are often allocated within the entire program-memory address space and fetched via the `lpm` command, which costs 3 clock cycles to execute.
Moving on to SRAM, the lower 224/352/607 data memory locations (bytes) address both the register file, the I/O memory and the internal data SRAM. The first 32 locations address the register file, the next 64 locations the standard I/O memory, and the last 128/256/512 locations address the internal data SRAM. We note that the SRAM is used for easy access of relatively small amount of information compared to flash memory; if certain data fits into SRAM, it can be stored there on-the-fly and be re-used, altered or deleted in the future. Several instructions access the SRAM (`ld,st,ldd,lds,sts`). They all manage access with the same time frame (2 clock cycles).
Finally, the ATtiny25/45/85 contains 128/256/512 bytes of data EEPROM memory, which is slower compared to SRAM and flash memory and will not be examined in our current line of work.

2 The PRESENT cipher

The first section of this work aims at suggesting a new bitsliced PRESENT cipher implementation that achieves high throughput performance, namely *2.9*× the throughput of the fastest non-bitsliced implementation (Papagiannopoulos, Verstegen [50, 49]) and *2.1*× the throughput of the fastest bitsliced implementation to our knowledge (Rauzy, Guilley, Najm [53]).
We commence by introducing the PRESENT cipher and its components and we continue by providing sufficient motivation towards a bitsliced approach (sections 2.1,2.2,2.4). We provide extensive insight to all PRESENT components inside the bitsliced implementation (section 2.5) and conclude with the performance metrics (section 2.6).
We use the following methodology: we offer a step-by-step examination and rebuttal of implementation choices, reaching the choice that we deem optimal. Note that this method is intuition-based and does not guarantee optimality in the strict sense.

2.1 Introduction

PRESENT [10] 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 it is named as a reference to Serpent [4] due to the similarity between the constructs. As of 2012, PRESENT (among other ciphers) was adopted by ISO as a standard for a lightweight block cipher (ISO/IEC 29192-2:2012 [2]). The full algorithm has so far been resistant to attempts at cryptanalysis, although the most successful attack has shown that up to 15 of its 31 rounds can be broken with $2^{35.6}$ plaintext-ciphertext pairs in 2^{20} operations [3, 17, 48].

Algorithm outline

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. Constructs found in PRESENT are also encountered in SPONGENT [9], in hash function constructs based on block ciphers as proposed by Hirose [11, 35, 36] (H-PRESENT) and in the similar Maya [28] or generalized SMALLPRESENT [42]. Thus the optimizations presented here can also be of interest with respect to the implementations of these ciphers. In our approach, we

have implemented PRESENT for the recommended 80-bit key size on AVR in bitsliced mode, attempting to achieve increased throughput.
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 (XOR) 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, while the permutation layer re-arranges the bits in the state following a 4-bit period. 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 the round key is XORed one last time (see also Figure 3).

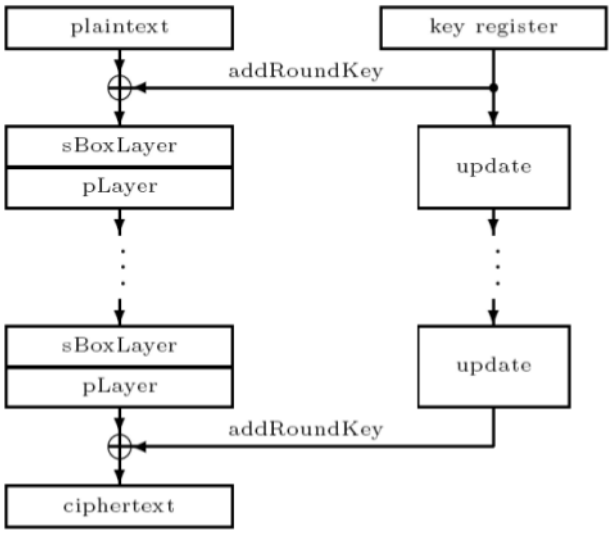

Figure 3: Schematic of the PRESENT cipher. It consists of 31 rounds, including XOR round key application, nibble-wise substitution, bit position permutation and key update.

Cipher components

`addRoundKey`. Given round key $K_i = k^i_{63}, k^i_{62}, \ldots, k^i_0, i \in \{1, 2, \ldots, 30\}$, round number i and current state $b_{63}, b_{62}, \ldots, b_0$ the add round key component consists of the following XOR operation:

$$b_j = b_j \oplus k^i_j, \forall j \in \{0, 1, \ldots, 63\}$$

`sBoxLayer`. The S-box used in PRESENT is a 4-bit to 4-bit S-box S: $GF(2)^4 \rightarrow GF(2)^4$. We display the substitution rules in the following table.

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 1: The original S-Box of the PRESENT cipher.

`pLayer`. The bit permutation used in present is given by by table 2. Bit 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

Table 2: PRESENT bit permutation table.

`keySchedule`. PRESENT can take keys of either 80 or 128 bits, yet our implementation is focused on 80-bit keys. Note that this does not largely affect our implementation choices (for 128 bit schedule see Bogdanov *et al.* [10], Appendix I). The user-supplied key is stored in a key register K and represented as $k_{79}k_{78}\ldots k_0$. At round i the 64-bit round key consists of the 64 leftmost bits of the current contents of register K (namely bits $79, 78, \ldots, 16$). After extracting the round key, register K is updated below:

1. $k_{79}k_{78}\ldots k_1k_0 = k_{18}k_{17}\ldots k_{20}k_{19}$
2. $k_{79}k_{78}k_{77}k_{76} = S[k_{79}k_{78}k_{77}k_{76}]$
3. $k_{19}k_{18}k_{17}k16k_{15} = k_{19}k_{18}k_{17}k_{16}k_{15} \oplus roundCounter$

2.2 Implementation motivation

As we have demonstrated in the previous section, the PRESENT cipher (and most ciphers based on substitution-permutation networks) employ a permutation layer to achieve cryptographic resistance versus known attacks. Specifically, PRESENT uses solely a *bit permutation* operation for the linear diffusion layer to achieve hardware efficiency, since bit permutations are simple re-wirings in hardware and increase the area only when the implementation is taken to the place-and-route step. Thus, it avoids the less hardware-friendly AES-like diffusion techniques such as the MixColumn operation [20]. The PRESENT permutation layer was crafted to provide resistance to differential cryptanalysis by ensuring that any five-round differential characteristic of present has a minimum of 10 active S-boxes. This is achieved with the bit permutation structure described in Figure 4, where the 16 S-boxes are divided into four groups while maintaining the following properties: *a)* the input bits to an S-box come from 4 distinct S-boxes of the same group, *b)* the input bits to a group of four S-boxes come from 16 different S-boxes, *c)* the four output bits from a particular S-box enter four distinct S-boxes, each belonging to a distinct group of S-boxes and *d)* the output bits of S-boxes in distinct groups go to distinct S-boxes (see [10], Section 5.1). The resulting pattern of the permutation is the following:

For i, j : *old, new position*,

$$j = f(i) = 16 \cdot (i \bmod 4) + (i/4) \tag{1}$$

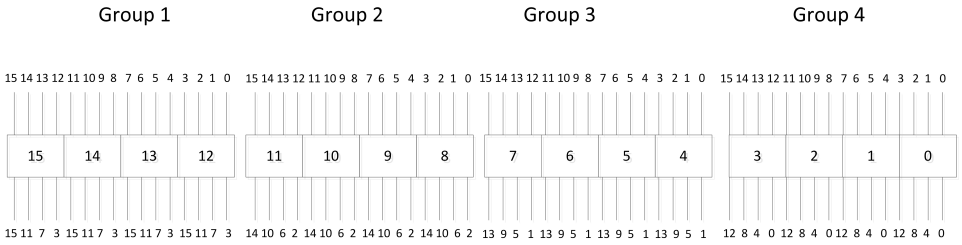

Figure 4: PRESENT cipher permutation layer. The layer ensures resistance *vs.* differential cryptanalysis by maintaining the afore mentioned properties *a,b,c,d*. It is bit-oriented and can be efficiently implemented with hardware wirings.

Having described the distinct hardware characteristics of PRESENT, we proceed with our motivation towards a software implementation that takes into account the cipher's structure. Obviously, we can implement the permutation layer naively, using in-place rotations, shifts and bitwise logical operations resulting in a very compact yet inefficient implementation (as described by Verstegen and Papagiannopoulos [50], Section 4). However, two factors will largely limit our throughput capabilities.

- The high structural complexity of the permutation layer, stemming from the design choice to optimize the cipher's hardware size, while sufficiently retaining resistance *vs.* differential cryptanalysis. The PRESENT bit permutations are efficient hardware wires and not particularly suited for software, unlike *nibble permutations* as encountered in AES and PRINCE shiftRows operations [20, 12].

- The low performance of the AVR architecture when computing permutations naively, *i.e.* when computing an operation which contains a large number of shifts or rotations. Specifically, calculating an *n-bit shift*, requires *n clock cycles* in an AVR microcontroller, creating a performance bottleneck. To overcome this, Hutter and Schwabe [38] suggested the usage of multiplication instructions in ATmega microcontrollers, *i.e.* multiplying by a constant to achieve shifting (costing 2 clock cycles per rotation/shift). However, since we solely focus on the ATtiny architecture, multiplication instructions are not available, hence we explore different alternatives.

We also note that there exists ongoing research that attempts to reduce the computational cost inflicted by software permutations. Permutation as a diffusion method is widely used in cryptography, thus the need to add general-purpose permutation instructions to either general-purpose processors or application-specific cryptographic coprocessors. The desired effect is that processors are capable of executing cryptographic functions at network link speeds, or at least with insignificant performance degradation. So far, permutations were usually examined from the scope of fast processing of digital multimedia information, using subword-parallel instructions (multimedia instructions [44, 43]), since multimedia processing required permutations on subwords of typically 8 bits or 16 bits. However cryptography (*e.g.* PRESENT cipher) may require permutations on bit level, tailored for 64-bit blocks.
Under this motivation, Ruby Lee *et al.* [45, 55], suggested extension to existing instruction sets to compute arbitrary *n-bit* permutations. Unfortunately,

we have not encountered such extensions in the AVR ATtiny architecture, with the exception of the efficient yet narrow-focused AVR DES instruction (an extension to the AVR ISA consisting of a built-in implementation of the DES cipher, costing 1 clock cycle per DES round). Still, we maintain that future effort in the area can result in *both fast and compact* cipher implementations, resolving the major cipher implementation hindrance that we have identified in AVR.

To our knowledge, the only general method to compute bit-level permutations in AVR is via the `bld, bst` instructions which load/store one bit a time from/to a specific register position in conjunction with register flag T, resulting in $2n$ clock cycles for an n-bit permutation (similar to the EXTRACT/DEPOSIT instructions described by Lee [44, 43]; note that the traditional approach to EXTRACT/DEPOSIT without `bld,bst` for direct bit addressing costs $3n + shift$ cycles per bit permutation). The resulting performance for the PRESENT cipher is very efficient memory-wise (no additional memory or registers are required) but it amounts to $2 \cdot 64 = 128$ clock cycles for a single 64-bit permutation and a total of $31 \cdot 128 = 3968$ clock cycles for a full PRESENT encryption, which is subpar compared to both the existing lookup table approach [50] and to the bitsliced method presented in this work.

	Number of instructions	**Memory**	**Total cost**
`mask`	$3n + shift$	0	ca. 12K cc
`bld,bst`	$2n$	0	3968cc

Table 3: Instruction-set-based approaches to permutation implementation. Both techniques implement EXTRACT/DEPOSIT functionality, albeit AVR ISA provides `bld,bst` to increase performance. Memory requirements are minimal, yet in section 2.5 we discuss that the total permutation cost (3rd column) via `bld, bst` is greater than that of the bitsliced approach.

2.3 Lookup tables for the PRESENT cipher

There exists a fairly wide spectrum of implementation choices with respect to latency, throughput and code size. For the PRESENT cipher, the first attempt to tackle the low performance of ATtiny regarding permutations was to fully replace them with lookup tables. The naive approach would be the following: to achieve a fixed permutation of n input bits with one

table lookup, we would need a table with 2^n entries, each entry being n bits. For a 64-bit permutation, this method would require 2^{67} bytes, an infeasible number for high-end systems, let alone AVR microprocessors.
Bishop [8], suggested an alternative method in order to compute permutation lookup tables for the DES algorithm. The technique presents structural similarities to programmable logic arrays (PLA), albeit from a software perspective. Namely, in this method, the source is divided into several sections. Then, the bits in each section are permuted simultaneously by looking up a table and finally, we combine the result of each section to obtain the result of the permutation. The number of instructions in this method is dependent on how many sections we divide the source into fewer sections require fewer instructions but more memory. For instance, a 64-bit permutation can be carried out as follows: divide the source into 8 sections, and build 8 tables where each table has 2K bytes (256 entries and 8 bytes in each entry). Now, only 8 bits are permuted within a single lookup, each table entry has 64 bits, and bits not permuted with that table are set to 0. In order to compute the final result, we need OR instructions that recombine the lookups (Figure 5). This line of approach is not unfeasible in a modern, high-end processor chip with a 64-bit architecture. Still, the AVR ATtiny uses an 8-bit architecture and since each table entry consists of 64 bits, we would require 8 AVR flash memory lookups for a single table lookup, which will largely diminish any speed benefits of this method. In addition, the size of the lookup tables is fairly large and requires 16KBytes of flash storage, a size that not even the newest ATtiny1634 can handle.

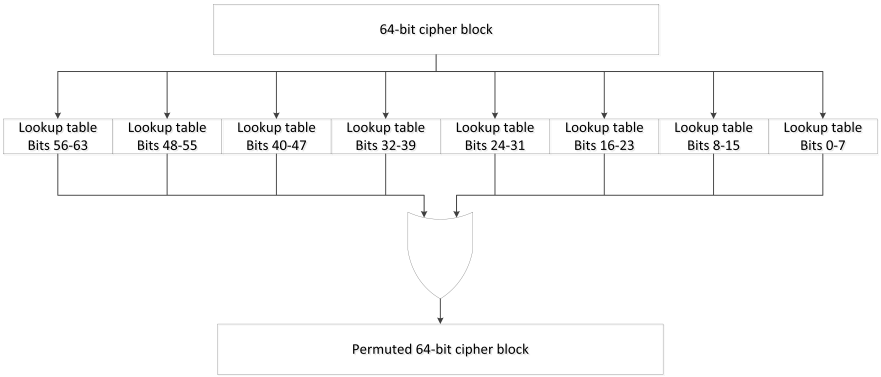

Figure 5: Permutation layer, using partitioned lookup tables. Each table computes 8 bit permutations and sets the remaining 56 bits to zero. The partitioned lookups are recombined via OR operations.

The best implementation of PRESENT lookup tables that we have identified so far is based on the observations by Gong and Zhu [29, 31]. As described in section 2.2, a property of the permutation layer of PRESENT is the following: every output of a 4-bit S-box will contribute one bit to the cipher; the first 2 bits of the output are derived from the first two 4-bit S-Boxes, *i.e.* from the first *byte* of the previous state (see Figure 6). Since every input byte contributes 2 bits to the output, we have the basis to form byte-oriented lookup tables for the PRESENT permutation layer. Using this basis, Gong and Zhu crafted four 256-byte lookup tables (1024 bytes in total) that merge the S-box and the permutation layer and as a result, the whole PRESENT SP network is performed via table lookups.

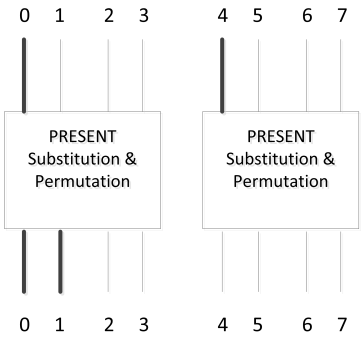

Figure 6: Zoom-in to the two leftmost nibbles (half-bytes) of the PRESENT 64-bit block. According to the permutation pattern, input bits 0,4 (highlighted in bold) get substituted-permuted and result in output bits 0,1 (also highlighted in bold). Concluding, a single input byte, contributes 2 bits to the output, setting the basis for 8-bit-based lookup tables.

This approach is also feasible in the AVR ATtiny architecture and was discussed by Papagiannopoulos and Verstegen in [50]. Performance-wise, the 1024 byte lookup tables eliminate the need for an independent permutation layer, providing us with a fairly fast and low-latency performance, namely 8721 clock cycles if the lookup tables are stored in flash memory, or 7729 clock cycles if tables are stored in SRAM, where the slow load from flash, using `lpm` instruction (3 clock cycles) is replaced by the faster SRAM load instruction (`ld`, 2 clock cycles). We note that this is feasible only in ATtiny1634, the only AVR ATtiny microcontroller with 1KByte of SRAM to this date. Another downside of the technique is the large number of memory accesses; a 64-bit block requires 32 memory lookups (each lookup

returning 8 bits), limiting the throughput.

2.4 The bitslicing technique

As discussed, the large number of table lookups, combined with the large storage space required for them, led to different approaches towards performance.
Bitslicing was first introduced by Biham [7] in order to improve the performance of bit permutations in software. We note that there exist structural similarities between DES and PRESENT; although DES is a Feistel network instead of an SP network, both are hardware-oriented ciphers that rely heavily on bit permutations which are efficient with wirings, yet slow in software. Bitslicing views a 64-bit processor (*i.e*, a processor capable of manipulating 64-bit operands) as a SIMD[4] computer with 64 single-bit processors (*i.e.* 64 processors, running in parallel, manipulating one bit at a time).
A bitsliced DES implementation uses a different representation: the cipher block is not represented as a single 64-bit register. Instead, it uses 64 registers, consisting of 64 bits each, in order to represent 64 independent cipher blocks consisting of 64 bits each. To clarify, we use the notation bit_j^i, where i indicates the cipher block ($i \in I = 0, \ldots, 63$) and j indicates the bit number ($j \in J = 0, \ldots, 63$), for instance bit_7^3 represents bit in position 7 of the block 3. Thus, in our representation, register 0 contains bit 0 of all 64 cipher blocks, register 1 contains bit 1 of all 64 cipher blocks *etc.*

Register 0	**Register 1**	**...**	**Register 63**
$bit_0^0, bit_0^1, \ldots, bit_0^{63}$	$bit_1^0, bit_1^1, \ldots, bit_0^{63}$	...	$bit_{63}^0, bit_{63}^1, \ldots, bit_{63}^{63}$

Table 4: Bitsliced representation of cipher block in DES algorithm, Each register stores a single 'bit', *i.e.* 64 bits from 64 different blocks representing the same conceptual bit position.

As a result, the DES expansion and permutation do not cost any operations, since instead of changing the positions of several bits within a 64-bit register, we simply reorder or rename the registers. For instance if we need to permute bit 6 to bit 27 we can simply address register 6 as register 27.

[4]Single instruction, multiple data (SIMD), is a class of parallel computers in Flynn's taxonomy [25]. It describes computers with multiple processing elements that perform the same operation on multiple data points simultaneously.

Every time we permute a single bit, we permute in fact 64 bits in parallel (hence the SIMD nature of bitslicing).

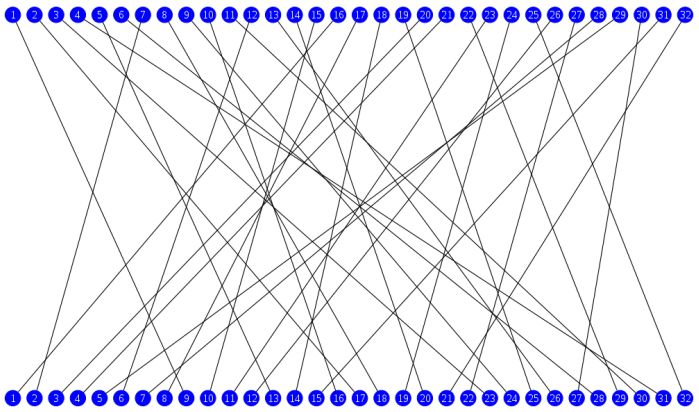

Figure 7: DES permutation operation. Despite being different to PRESENT Player, structural similarities exist since both are bit-based, hardware-oriented permutations. Under bitslicing, the whole operation is reduced to mapping registers,*e.g.* register 1 $\rightarrow$ register 9, register 2 $\rightarrow$ register 17 *etc.*

Note that two factors define the range of the bitslicing parameters i, j:

- The number of bits j is directly determined by the size of the cipher block we are trying to manipulate. In the case of DES, we need bit permutations on a cipher block of size 64 bits, hence we use 64 bits in our representation. If the cipher used 32 bit blocks, or if the permutation consisted of two symmetrical parts, *i.e.* the permutation on the 32 low bits was same with the permutation on the 32 high bits, then number of bits $j \in J = 0, 1, \ldots, 31$.

- The number of blocks i that are permuted in parallel is related to the processor architecture. A 64-bit processor can manipulate 64 bits in parallel if viewed as a SIMD computer, thus Biham suggests 64 DES blocks being computed in parallel. Parameter i is often called the 'bitslice factor' and defines the upper bound of parallelism when using bitslicing. A 32-bit processor can have maximum parallelism of 32 blocks, while the AVR ATtiny (with an 8-bit architecture) can process 8 cipher blocks in parallel.

In general, we can apply the following formulas when tailoring bitslicing for a cipher in a given platform:

$$Max\ parallelism = |I| = register\ size^{5} \tag{2}$$

$$Registers\ used = |J| = cipher\ block\ size \tag{3}$$

The bitsliced DES implementation (or any bitsliced cipher implementation in general [54]) presents several limitations that can can lead to applications where bitslicing excels over traditional encryption. For intance, it can be used to encrypt *large* plaintext messages, *i.e.* encrypting 64 plaintext blocks in parallel with the same key (if using a parallel mode of operation). In addition, Biham points out that alternating plaintext and keys, namely encrypting a single plaintext with multiple keys, can produce a bitsliced exhaustive key search. As discussed, the permutation layer of ciphers is largeley simplified and increased throughput is achieved.
However, despite the advantages of bitslicing with respect to the permutation layer of ciphers, we also encounter several issues. The substitution layer becomes more complex due to the fact that we can no longer use lookup tables under bitsliced representation, forcing us to either un-bitslice temporarily or implement the S-boxes via software boolean functions. Moreover, bitslicing can limit our options of cipher modes. While CBC[6] decryption mode can be carried efficiently in parallel, the sequential nature of CBC, CFB[7] and OFB[8] encryption, poses hindrances to this technique, requiring 64 parallel CBC encryptions to maintain the parallel SIMD potential ([7], Section 2, page 6). Last but not least, we must always note that any bitsliced cipher implementation is inherently a *high-latency* implementation, since it carries out several encryptions in parallel. Thus, it may be rendered inviable in high-speed, low packet size network scenarios.

[5] Note that by register size we mean the size which is directly operable by the processor, excluding certain architectures where instructions operate on register sub-components

[6] Cipher-block chaining.

[7] Cipher feedback.

[8] Output feedback.

2.5 Bitslicing the PRESENT cipher

In this section we offer a full insight into all the implementation details of the bitsliced PRESENT cipher. Bitslicing this cipher has been attempted before both in C language(Albrecht *et al.* [47]) and AVR assembly (Rauzy, Guilley, Najm [53]). We commence with an in-depth exploration of the cipher's S-box under the bitsliced approach (sections 2.5.1,2.5.2). We continue with the permutation layer (section 2.5.3) and conclude with sections pertaining to key precomputation, update and *XOR* operation (sections 2.5.4,2.5.5).

2.5.1 Substitution layer under bitslicing

Assuming 4-bit S-boxes, a cipher block size of 64 bits and an 8-bit architecture (these parameters pertain to the PRESENT cipher in AVR ATtiny architecture), performing a substitution directly via lookup tables becomes impossible. With a bitsliced representation, we need to perform a lookup based on 4 registers (instead of 4 bits), *i.e.* $4 \cdot 8 = 32$ bits, which is not possible given *a)* the 16-bit addressing capabilities of AVR ATtiny (the `lpm` instruction operates using a 16 bit address in registers `ZH, ZL`) and *b)* given the fact that the resulting lookup table would be infeasible in size.
A more viable alternative would be to first *extract* the bits required out of the bitsliced representation, in other words, temporarily revert to the original form, perform a lookup and then store back in the bitsliced representation. However, there exists a large performance overhead when performing such an operation: un-bitslicing 8 bits from 8 registers (*e.g.* extracting bit_0^4, bit_1^4, ... , bit_7^4), performing a lookup based on those 8-bits (using the AVR-friendly squared S-boxes [50, 23]) and restoring the substituted bits back to the bitsliced representation will cost circa 20 clock cycles[9], thus $20 \cdot 64/8 = 160$ clock cycles for a single block and 1280 clock cycles for 8 blocks (8 being the bitslice factor). The technique is visible in Figure 8 .

[9]Note that this holds if we perform multiple 8-bit extractions, lookups and reconstructions; we also need to interleave rotations that input bits to `ZH,ZL` and rotations that output bits back to the registers.

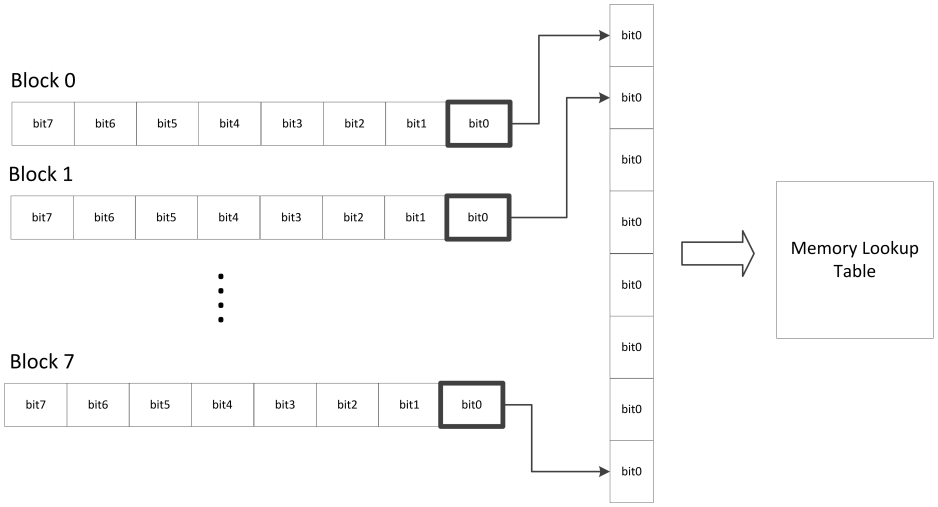

Figure 8: Temporarily un-bitslicing the representations to perform memory lookups. Performance is circa 1280 clock cycles for 8 blocks.

The best solution that we have identified so far for computing efficiently the substitution layer of a cipher in bitsliced representation is by viewing the S-box as a boolean function. Bitslicing is essentially a simulation of hardware operations in software, and it has partitioned the representation into 'bits' (more correctly: registers). When implementing any boolean function under bitslicing, we still maintain the SIMD parallelization – unlike the approach based on temporarily un-bitslicing. For instance, performing a binary AND operation between two AVR registers, performs the operation with 8-bit operands simultaneously (see Figure 9). We discuss software boolean functions more extensively in the following section and we produce an implementation that computes the substitution layer of PRESENT cipher in 304 clock cycles per 8 cipher blocks.

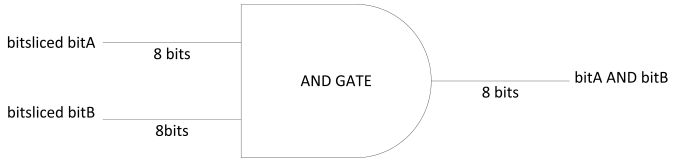

Figure 9: SIMD parallelization is maintained when computing a logical expression. The AND gate represents the instruction `AND bitA, bitB`, where `bitA, bitB` are bitsliced registers.

2.5.2 Efficient software implementation of boolean functions

In section 2.5.1 we have established a strong argument of implementing the bitsliced cipher's S-box using the equivalent boolean function in software. We henceforth refer to boolean functions implemented in software as 'software boolean functions' and to boolean functions implemented in hardware as 'hardware boolean functions'.
In order to efficiently implement a software boolean function we point out its close resemblance to hardware construction of optimal circuits; in fact, we will demonstrate that software boolean function implementation 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.*). It interfaces with NP-complete algorithmic problems such as set cover and best variable ordering in binary decision diagrams [18]. 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.
The first crucial step of hardware (and respectively software) boolean function construction is *logic minimization* and *decomposition*, *i.e.* the process of finding an equivalent representation of the specified logic circuit under one or more specified constraints - usually area or delay - and translating the logic network into some primitive cells to create a simple structure that aids the whole mapping process.

Boyar-Peralta heuristic

In 2008, Boyar and Peralta introduced an efficient new heuristic methodology to minimize the complexity of digital circuits [13, 14, 1], with the focal point being efficient cipher implementation. The heuristic is based on the notion of *Multiplicative Complexity* (MC) which is a deep fundamental notion of complexity invariant with respect to affine transformations. More specifically, Boyar and Peralta attempted to minimize the logic of boolean functions over the logically complete basis $\{\oplus, \wedge, 1\}$, *i.e.* the circuit operations can be viewed either as performing boolean logic or arithmetic modulo 2. They define the multiplicative complexity of the circuit being the number of AND ($\wedge$) gates and decompose the circuit into *linear* components (not containing $\wedge$), and *non-linear* components (containing $\wedge$). The suggested heuristic is based on the following conjecture:

*"it is plausible that a two-step process, which first reduces multiplicative complexity and then optimizes linear components, leads to small circuits, since circuits with low multiplicative complexity will naturally have large sections which are purely linear (*i.e. *contain only* $\oplus$ *gates)."*

Boyar and Peralta applied the heuristic to the AES substitution layer (modeled by an 8-bit to 8-bit boolean function), first by reducing the multiplicative complexity (step 1) and subsequently by minimizing the linear components (step 2), resulting in a very efficient hardware implementation of AES S-boxes, consisting solely of AND and XOR gates.

Courtois extension and application of Boyar heuristic

Courtois, Hulme and Mourouzis [19] extend this conjecture and apply the heuristic to several S-boxes modeled by $GF(2)^4 \rightarrow GF(2)^4$ boolean functions (including the PRESENT cipher S-boxes), which are particularly interesting for lightweight cryptography (PRESENT [10], PRINCE [12], KLEIN [30],*etc.*). We continuing by discussing *a)* the additional definitions/premises upon which they construct their heuristic, *b)* the methodology they introduce and its capabilities and *c)* the results that are of interest with respect to the bitsliced PRESENT implementation that we have constructed for the AVR platform.

Bitslice Gate complexity

In addition to the existing multiplicative complexity metric, Courtois *et al.* introduce the notion of *Bitslice Gate complexity* as the minimum number of 2-input gates of types XOR,OR,AND and single-inpute 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. However, at this point *we are able to 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.
To compute minimal boolean representations of the PRESENT S-boxes, Courtois *et al.* use multiplicative complexity instead of bitslice gate complexity as the heuristic, due to the fact that with Bitslice Gate Complexity we are not in general able to determine its value, algorithms which find such optimizations are typically random stochastic explorations of large trees of solutions [27] and we are not sure if the optimizations are final or if they can still be improved. However, as we will show in the next paragraphs, after using the Boyar-based heuristic, Courtois attempts to minimize the bitslice gate complexity of the PRESENT S-box via affine transformations of gates. This is not directly applicable to hardware due to the underlying assumption of gate cost equivalence (*i.e.* all gates cost the same), it is of particular interest for a bitsliced software approach that can employ equivalent XOR,OR,AND,NOT instructions.

Courtois methodology and optimality

Another point of particular interest is the efficiency of the Courtois methodology and to which extent the heuristic yields optimal results for PRESENT. An important assistance to optimality is the fact that we deal with $GF(2)^4 \rightarrow GF(2)^4$ S-boxes, hence 4-to-4 boolean functions. Finding the optimal representation of such functions is not computationally infeasible with exhaustive search, despite the general intractability of the problem. The methodology used in PRESENT S-boxes is the following:

1. Compute the multiplicative complexity and minimize the non-linear parts [13].
2. Optimize the number of XORs (linear parts) separately [13, 26].
3. At the end do additional optimizations to decrease the circuit depth, and possibly additional software optimizations.

Steps 1 and 2 are similar to the Boyar methodology and both are *optimal*. Optimality is achieved due to SAT solver software; Courtois *et al.* converted the problem to SAT and it either outputs SAT[10] and a solution, which they

[10]SAT is the problem of determining if there exists an interpretation that satisfies a given boolean formula. In other words, it establishes if the variables of a given boolean formula can be assigned in such a way as to make the formula evaluate to TRUE

convert to a concrete circuit optimization, or it outputs UNSAT[11], providing certainty that there is no solution. Although in general the SAT solver software may run for a very long time and the problem may be hard, the boolean functions are fairly small (4-to-4) so SAT decidability is always computationally feasible.
Steps 1 and 2 essentially result in an optimal representation using the basis $\{\oplus, \wedge, 1\}$, yet not optimal with respect to the bitslice gate complexity metric. Step 3 uses an additional heuristic: it observes that AND gates and OR gates are affine equivalents and it is likely that if we implement certain AND gates with OR gates we might be able to further reduce the overall complexity of the linear parts. Thus, it employs exhaustive search alternating between AND and OR gates, resulting in a very efficient, yet not rigidly optimal implementation with XOR,OR,AND,NOT gates.

Results of the Courtois approach

The results of the approach discussed 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 [47]. Steps 1 and 2 result in a PRESENT S-box with 25 gates: 4 AND, 20 XOR, 1 NOR which is optimal with respect to the Boyar-Peralta 2-step methodology but not optimal in overall gate complexity. After the application of step 3, the representation becomes the following:

```
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 translating the afore mention`operator destination, source` instead of `operator destination, sourceA, sourceB`. The inherent result is that it is not possible to reuse an computed value, unless we store it temporarily elsewhere. For instance, we compute T_1 in the first step of the computation and we need to reuse it (as is) in steps 2 and 5, thus we need to store it temporarily. With careful register usage, we maintain this penalty to a minimum and our final implementation requires 19 clock cycles to compute the

[11] UNSAT is a boolean function that is unsatisfiable, *i.e.* no variable assignment exists to make the function TRUE.

output of a single PRESENT S-box (*i.e.* a penalty of 5 clock cycles). As a result, the 16 S-box operations used in the bitsliced representation require $19 \cdot 16 = 304$ clock cycles on 8 blocks in parallel.

2.5.3 Permutation layer under bitslicing

In general, the permutation layer under a bitsliced representation has zero cost. Unfortunately, that does not imply zero clock cycles. Since we are bitslicing a 64-bit cipher block with a bitslice factor of 8, we would naturally require 64 registers (8 bits each) and permutation would result in simple register renaming or reordering. However, the AVR ATtiny architecture provides us with 32 8-bit registers, rendering this impossible and forces us to use the SRAM for storing the cipher state, whose size is $64 \cdot 8$ bits= 64 bytes. The implementation we propose uses the following iteration:

1. Fetch 4 bitsliced 'bits' from SRAM to the registers, *i.e.* transfer 4 bytes, each representing a 'bit', to the registers.
2. Use the conclusions of the Courtois approach in the previous section (2.5.2) to compute the 4 byte output of the S-box, also in bitsliced form.
3. Store the substituted 'bits' back to SRAM. In this step, we integrate the PRESENT permutation inside the memory stores. Integrating the permutation is not de facto required but storing without any consideration for the permutation would require us to hard-code all 31 rounds of PRESENT in order to implement the permutation logic.

We provide an example (also visible in Figure 10). Bitsliced 'bits' 0,1,2,3 are fetched from SRAM and get substituted via the S-box boolean function computation. Subsequently, they are stored back to SRAM in permuted form, namely 'bit' 0 gets stored in the position of 'bit' 0, 'bit' 1 in the position of 'bit' 16, 'bit' 2 in the position of 'bit' 32 and 'bit' 3 in the position of 'bit' 48, according to the PRESENT cipher permutation pattern explained in section 2.2. Note that we have to repeat the exact same procedure for all the 16 'bit'-quartets that construct the 64-'bit' cipher block and the only way to do this is sequentially. Thus, after we have fetched and substituted the first 'bit'-quartet, we have to store in positions 16,31,48 which would overwrite the non-substituted values stored there. To avoid this we increase the used memory space from 64 bytes to 128 bytes. Essentially, we partition those 128 SRAM bytes in two. The first partition contains all the 'bits' we

are going to fetch, while the second partition is free to store all permuted 'bits' according to the pattern that we want. For example, in round i the implementation will fetch from the first partition, substitute and store to the second partition in a permuted fashion. In round $i+1$ the roles alternate: we fetch from the second partition and store to the first partition in a permuted fashion.
It is important to know that the memory transaction cost is non trivial; every single round of the bitsliced PRESENT cipher will require 64 memory fetches and 64 memory stores.

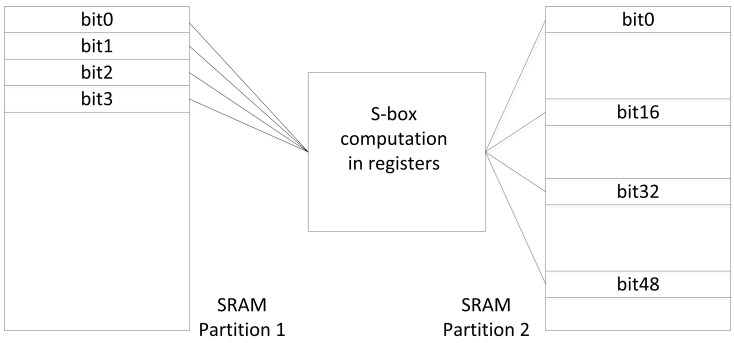

Figure 10: A running example of the implementation. From left to right: four 'bits' (more correctly: 4 bytes) are fetched from the SRAM partition 1, the boolean function is computed and the result is stored to the SRAM partition 2, including the required permutation. In the next round the procedure remains similar albeit it executes from right to left and the partitions' functionality is exchanged.

Efficient memory accesses

Having demonstrated that the computational cost of the permutation layer is not trivial in our architecture (since *a)* we need memory to store the bitsliced state and *b)* we need to integrate the permutation in the storage function), we proceed by identifying the optimal way of constructing it.
Regarding the memory fetch from partition 1, we observe that all required memory accesses are aligned linearly. Thus, we can use the instruction `ld destination, X+`, where `X` is the starting address of partition 1 and '+' denotes that the starting address is linearly increased by 1 after the instruction, giving access to the next byte (or bitsliced 'bit') in partition 1. The memory access cost is 2 clock cycles which we identify as the minimum cost when accessing the AVR ATtiny SRAM, in the sense that no other instruc-

tion can access SRAM with less than this threshold.
The same technique will not work efficiently for storing bytes in SRAM partition 2 in permuted fashion. The access pattern is well-defined, yet it requires memory addressing that begins from the starting address of partition 2 and has offset $i \in \{0, 1, \ldots, 63\}$, where i denotes the 'bit' position that will be accessed and i is not sequential. In order to maintain the cost at 2 clock cycles, we avoid the usage of `ldi YL,i` and subsequently `st Y,destination`. Instead, we can opt to either use direct memory addressing via the `sts` instruction or store indirect from register to data space using index Y, via the `std` instruction.

2.5.4 Key precomputation and update under bitslicing

A standard technique for increasing throughput and decreasing latency in cipher implementations consists of precomputing the key for all cipher rounds and storing it in memory for fast access. Although this often holds (especially when re-keying is rare), in the bitsliced case the technique becomes less practical.
Storing 31 keys in bitsliced form requires $31 \cdot 80 = 2480$ bytes for the 80-bit key version of PRESENT, thus it can only be stored in flash memory. Although the size is manageable, a bitsliced key needs to be XORed with the cipher state every round, while fetching each bitsliced 'bit' costs 4 clock cycles (`ldi ZL,position; lpm register,Z`) and in total $64 \cdot 4 = 256$ clock cycles per round[12], a considerable overhead. The advantage of this method is that if applied, it can achieve encryption with multiple keys, namely each block can be encrypted with a separate key, however this server-like scenario is not often encountered in AVR applications - usually there is a single client to communicate with, using a single shared key.
The option left is to store/use the key in a non-bitsliced form. We can either opt to perform precomputation or perform the key update at runtime.

- If we choose to precompute, we need to store $80 \cdot 31 = 2480$ bits or 310 bytes which cannot be stored in SRAM unless we use a device with at least 512 bytes of SRAM (ATtiny8x or above), given the fact that 128 SRAM bytes are also demanded by the permutation layer. Since we only need to fetch 64 bits per round, the cost is $8 \cdot 2 = 16$ clock cycles if fetching from SRAM or $8 \cdot 4 = 32$ clock cycles if fetching from flash memory, resulting in a total cost of either $31 \cdot 16 = 496$ (SRAM) or

[12]Note that we do not need to fetch the whole 80 'bits' since only 64 are being used per round.

$31 \cdot 32 = 961$ (flash) clock cycles in total.

- If we do not precompute the key values, we need to perform the key updates while iterating the PRESENT cipher. The cost amounts to 61 clock cycles per round, or $31 \cdot 61 = 1891$ in total. Our implementation currently uses this approach.

Efficient key update

This subsection focuses on implementing the key scheduling/update process of the PRESENT cipher efficiently. The key update function of the PRESENT cipher consists of three operations, namely, key rotation, key substitution and key XORing with the round counter. The optimizations described in the following paragraphs have also been demonstrated in the AVR context by Eisenbarth *et al.* [23] and by Verstegen & Papagiannopoulos [50]. We also present them here for the sake of completeness.
Key Rotation The cipher key must be rotated by 61 bits to the left. Having discussed the fact that rotations/shifts are computationally expensive in the AVR architecture (an n-bit shift costs n clock cycles), we transform 61 left rotations to 19 right rotations, which can be further reduced to 16 right rotations and 3 right rotations. The 16 right rotations can be efficiently implemented by using the `mov` instructions on register level (unless we use the less feasible hard-coding approach), *i.e.* rotate all the bits inside a register by moving the contents to the previous register used in our representation, an approach which is preferable to single rotations via the bit-level instructions. The 3 remaining rotations are carried out with the logical instructions for right rotation and shifting (`ror` and `shr`).
Key Substitution. The highest 4 bits of the 80-bit key used by the PRESENT cipher, must be substituted via the S-box. To avoid 4-bit memory access or redundancy, we use the special-purpose Squared S-Box [50] that substitutes the 4 high bits of the 8-bit input, while the low 4 bits remain unchanged. The resulting table applies a substitution operation on the upper nibble which takes only a single lookup operation. Note that if we have limited flash memory space, it is possible to replace the squared S-box with the original, unpacked one; the key substitution occurs only once per round, so the performance loss incurred by the unpacked S-box is relatively small.
Key Exclusive-OR Operation. The algorithm specifies that the key bits 15, 16, 17, 18, 19 must be XORed with the round counter. The issue is that—under the current non-bitsliced key representation—bits $0 \ldots 7$ will be stored in register0, bits $8 \ldots 15$ will be stored in R1 and bits $16 \ldots 23$ will be stored in

R2. As a result parts of the round counter need to be XORed with different parts of two separate registers, namely the counter needs to be XORed with both R1 and R2. To avoid this, we perform the XOR operation before the key rotation, thus the bits that are operated on are bits 34,35,36,37,38 which span a single register (under the previous representation they are located in R4). This form of register re-arrangement in order to group bit values together will also be used in order to improve the speed of the KATAN cipher (section 3).

2.5.5 Key XORing under bitslicing

The last implementation section focuses on the XOR operation between the cipher state and the cipher key. In a straightforward, non-bitsliced implementation the XOR step is fairly trivial, while under bitslicing it becomes more complex.
In the previous section (section 2.5.4), we provide a strong argument of storing the key in a non-bitsliced form (the argument pertaining to the increased cost of memory fetches). However, the cipher state employs the bitsliced representation, rendering it difficult to perform the XOR operation. Once again, it is possible to use the temporary un-bitslicing method (explained in section 2.5) to revert the plaintext back to traditional representation, XOR with key and reconstruct the bitsliced form, yet we have identified better alternatives.
We base our implementation on the following observation:

```
if ( key_bit_j == 1)

     XOR bit_j^i with 1;
     ∀ i ∈ {0, 1, ..., 63}
else

     XOR bit_j^i with 0;
     ∀ i ∈ {0, 1, ..., 63}
```

where key bit j denotes the jth bit of the key and bit_j^i denotes the bit at position j, related to block i.
The observation demonstrates that if the key bit at position i is equal to 1, we need to XOR every single element of the bitsliced 'bit' i with the value 1 or, in other words, XOR the register containing 'bit' i with the value $0xFF$. Similarly, if the key bit at position i is 0, we would need to XOR the register containing 'bit' i with the value $0x00$. Attempting to implement this via the

compare/jump commands of AVR is not efficient and additionally, it cannot exploit the fact that `else` clause does in fact nothing, since $x \oplus 0 = x$. Any attempt to remove the `else` clause to increase speed can result in serious time invariance that is directly related to the key values and should be avoided at all times.
Fortunately, the AVR ISA provides us with an instruction that can perform the `if-else` clause, while being resistant to timing attacks. The `sbrc` command tests a single bit in a register and skips the next instruction if the bit is cleared. Thus, it checks the value of the key bit i and if it is set, it executes the next instruction, *i.e.* the XOR with value `0xFF`. In case bit i is zero, `sbrc` parses the XOR instruction but does not execute it, running, thusly, in constant time regardless of the key value both in ATtiny and ATmega.

2.6 Performance

In this section we focus on measuring the attained latency, throughput and size of the suggested implementation. In addition, we provide comparison with other previous attempts to implement the PRESENT cipher on AVR architecture and we demonstrate that we achieve the highest throughput to date. Despite that, we must point out that *a)* the memory requirements are quite large, forcing us to develop for the ATtiny85 microprocessor and that *b)* the suggested implementation is bitsliced, thus it is automatically rendered less useful in low-latency application scenarios.

Component cost

The following table provides us with an overview of the costs of individual cipher components (substitution, permutation, key update). Note that there are two ways we can view the measurements. The second column demonstrates the cost of *component per round per block*, thus, the cost of the component after dividing with the bitslice factor of 8. The third column offers the component cost per round per 8 blocks, which reflects more accurately the actual round iteration.

Component	**Cost per block**	**Cost per round (8 blocks)**
S-layer	38	304
P-layer	32	256
Key update	61	61
Total (31 rounds)	2967	23736

Table 5: Component cost (in clock cycles) of the proposed PRESENT bitsliced implementation.

We can observe that the substitution layer has the highest computational requirement, although the permutations—which were expected to decrease rapidly via bitslicing—are on par with it. This stems from the fact that we are using SRAM to store the cipher state, which in turn, results from the complex bit permutations of the PRESENT cipher. While discussing the SP network of the PRINCE cipher (section 4), we will demonstrate that reducing the complexity of the pLayer can result in increased performance with alternative bitslicing-like methods.

Throughput

PRESENT implementation	Throughput (cc/block)	Bitsliced
AVR Crypto-Lib [52]	105796	no
Eisenbarth *et al.* [23]	10723	no
Papag. [50, 49]	8721	no
Rauzy *et al.* [53]	6473	yes
This work	2967	yes

Table 6: Throughput of PRESENT cipher implementations for AVR architecture, *i.e.*, clock cycles required for a single encryption round.

The suggested implementation manages to outperform all existing implementations with respect to throughput. Comparing this work with the non-bitsliced work by Eisenbarth, 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 per bit). Our 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 P-layer of Eisenbarth (154 cc per round) compared to ours (32 cc per round). Thus, despite the SRAM penalty on the P-layer and the lower performance on S-layer, we can outperform both Eisenbarth. We also outperform Papagiannopoulos and Verstegen due to the fact that they replace the SP network with lookup tables that result in a large number of flash memory accesses (1 access per 2 bits).
Comparing our bitsliced version with Rauzy *et al.*, 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 access optimizations and XOR operation techniques, we cannot identify the source of this speed-up, although we observe that the authors managed this performance at a lower size.

Latency

Unfortunately, our implementation does not fare well regarding latency. All bitsliced implementations perform inherently multiple blocks in parallel (equal to the bitslice factor), so in our case we perform 8 block encryptions in parallel 23736 clock cycles. Thus, we are outperformed by Papagiannopou-

los and Verstegen with 8721 clock cycles or circa 7700 clock cycles when using an AVR ATtiny with 1024 KBytes of SRAM.

Throughput-size ratio

This works aims at improving the throughput of the PRESENT cipher, thus it is not efficient size-wise. Still, we do maintain that a bitsliced implementation is less reliant on lookup tables than a traditional one; thus, it is possible to tweak it in a size-efficient manner, while preserving relatively high throughput and this intended to be part of future work.

PRESENT implementation	**Flash (bytes)**	**SRAM (bytes)**
Eisenbarth *et al.* [23]	1000	18
Papag. [50, 49] ATtiny45	1794	0
Papag. [50, 49] ATtiny1634	770	1024
Rauzy *et al.* [53]	1194	144
This work	3816	256

Table 7: Size of PRESENT cipher implementations for AVR architecture in bytes, pertaining to flash and SRAM. No stack memory is used in those applications.

Last, since we have established metrics for both throughput, size and latency, it is of particular interest to observe the trade-offs that exist between these three variables, in order to be able to tailor given requirements to a specific implementation. We provide two figures: The first figure (11) demonstrates the throughput-size trade-off, while the second (12) shows the throughput-latency trade-off. Apart from those metrics, we note that there exist very slow, yet highly minimized implementations for PRESENT in AVR by Papagiannopoulos and Verstegen [50].

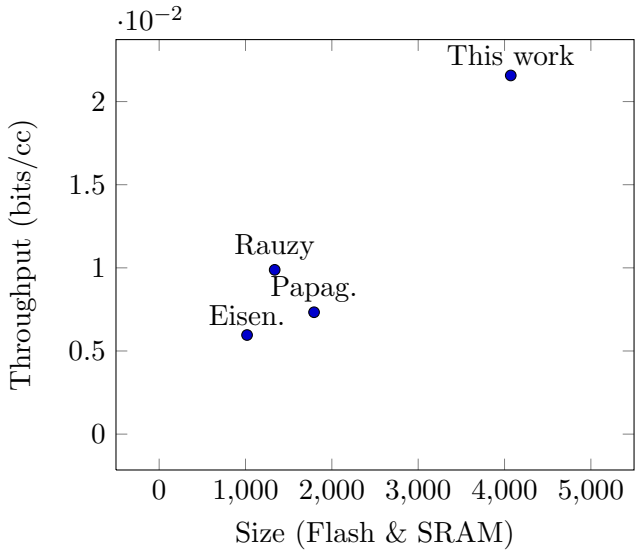

Figure 11: Throughput *vs.* Size diagram for various implementations.

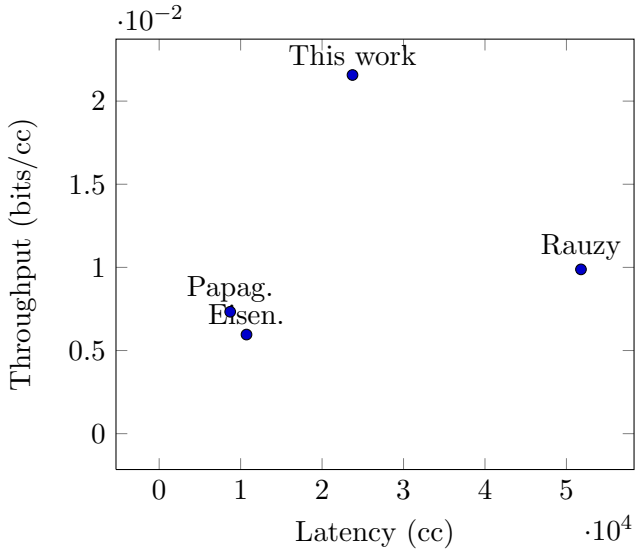

Figure 12: Throughput *vs.* Latency diagram for various implementations.

3 The KATAN cipher

The second section of this work examines a different type of cipher that is not related to SP networks and resembles a stream cipher. We begin by introducing the 80-bit keyed KATAN cipher (Section 3.1) and by motivating our implementation choices (Section 3.2). We proceed by proposing two novel ideas to improve performance (Section 3.3) and conclude with performance metrics (Section 3.4).

3.1 Introduction

The outline is provided in Figure 13. The KATAN cipher [15] was designed as a secure 80-bit block cipher with a minimal number of gates. Following the design of KeeLoq [24] the designers chose a structure similar to a stream cipher, resembling the two-register variant of Trivium [16], known as Bivium.

Algorithm outline

The cipher's plaintext is loaded into two linear feedback shift registers (LFSR) 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, thus the following paragraphs that describe the cipher's components pertain only to KATAN64.

Cipher components

`Non-linear functions` f_a, f_b `and bit rotation.` KATAN64 uses two non-linear function f_a and f_b in each round which are computed as follows.

$$f_a(L_1) = L_1[24] \oplus L_1[15] \oplus (L_1[20] \cdot L_1[11]) \oplus (L_1[9] \cdot IR) \oplus k_a \qquad (4)$$

$$f_b(L_2) = L_2[38] \oplus L_2[25] \oplus (L_2[33] \cdot L_2[21]) \oplus (L_2[14] \cdot L_2[9]) \oplus k_b \qquad (5)$$

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 implementation can implement these three steps in parallel, if it spends some extra gates.

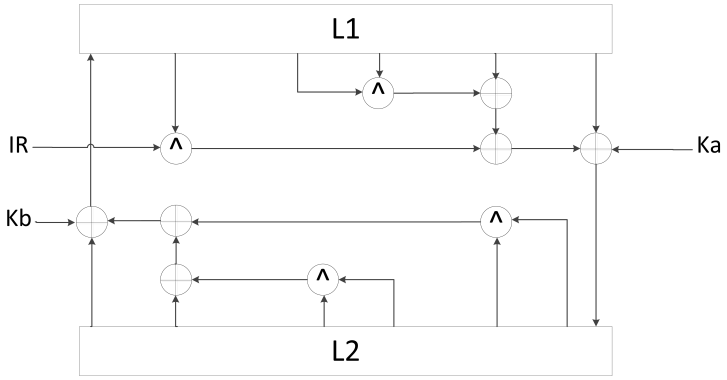

Figure 13: The core operation of the KATAN cipher. The two LFSR L1, L2 store the cipher state. Several bits are extracted from L1, L2, from the cipher key (k_a, k_b) and from IR in order to compute the non-linear functions f_a, f_b (via XOR/AND operations) and to update the cipher state.

`Key schedule.` The key schedule of the KATAN64 cipher loads the 80-bit key into an LFSR (the least signicant 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. The feedback polynomial that was chosen is a primitive polynomial with minimal hamming weight of 5 and is the following:

$$x^{80} + x^{61} + x^{50} + x^{13} + 1 \qquad (6)$$

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 $k_i = k_{i-80} \oplus k_{i-61} \oplus k_{i-50} \oplus k_{i-13}$.

3.2 Implementation motivation

Having discussed the construction of the KATAN cipher, we proceed by offering motivation towards a high-throughput implementation. To this date, the only implementation of the KATAN cipher for AVR devices is demonstrated by Eisenbarth *et al.* [23] and attempts to minimize code size by avoiding both flash and SRAM, instead of throughput. KATAN is a cipher optimizing for low-throughput and small hardware fingerprint: it uses a large number of single-bit operations and a large number of rounds (254), resembling the structure of a stream cipher. The large number of rounds, in combination with its bit-oriented structure, suggest that even small speed improvements on a single round can yield a large improvements overall.
In general, our implementation is motivated by two factors:

- The stream-like construction of the iteration function has the following side-effect: relatively small key size over the 254 rounds (especially when compared to *e.g.* PRESENT, which produces a different 64-bit subkey for each one of the 31 rounds). This observation can lead to key precomputation techniques which are not heavily reliant on memory accesses due to the low key size.

- 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 in our budget). 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.

3.3 Implementing KATAN cipher

In this section we provide a new KATAN64 implementation for AVR ATtiny architecture. We propose two ideas (sections 3.3.1 and 3.3.2 correspondingly) that increase KATAN64 throughput while keeping the cipher code size relatively small.

3.3.1 Key precomputation of the KATAN cipher

As discussed, the key schedule of the KATAN cipher (all three variants) loads the 80-bit key into an LFSR and during round i it generates the round's subkey k_{2i} and k_{2i+1}, while the LFSR is clocked twice. Every round needs 2 key bits for the iteration function and generates only 2 additional bits. The total key size used for a full 64-bit KATAN encryption is 508 bits, *i.e.* 80 bits of starting key and 248 of generated key bits (63 bytes in total). The key size is relatively small and unless re-keying is a very common procedure, the key can be precomputed in advance and stored in SRAM for consecutive future use. The low overall key size results in *a)* small number of memory fetches regarding the key and *b)* low key-storage requirements. Given the fact that AVR can fetch one byte from SRAM per load instruction (2 clock cycles), precomputing and fetching the key costs a total of 64 SRAM loads[13] ($508/8 \approx 64$ bytes), *i.e.* 128 clock cycles for a full 64-bit encryption. Note that such an approach was less attractive in SP networks. For instance, PRESENT requires 31 64-bit subkeys for a total of 248 key bytes stored in SRAM, which either demands a more expensive AVR ATtiny model (with 512 or more bytes of SRAM if we use the implementation suggested in chapter 2) or implies using flash storage which is slower roughly by a factor of 2, compared to SRAM. Similarly to the key, the irregular update sequence is also stored and fetched from SRAM.
The alternative to a precomputed key is computing the round key during the cipher iteration. Although the AVR architecture provides us with bit-oriented instructions such as `bld,bst`, key update will cost 9 bit extracts (`bld`), 9 bit deposits (`bst`) and 6 XOR operations per round, pertaining to at least $6096 = 254 \cdot 24$ clock cycles for a full encryption. As a result, key precomputation yields a large performance boost, albeit at the cost of increased SRAM usage (74 bytes).

[13] We assume that the original 80-bit key is already in place. If not, total cost becomes 148 clock cycles per full encryption.

3.3.2 Parallel bit operations on f_a, f_b non-linear functions

As discussed in section 3.2, KATAN contains 3-way parallelizability when computing the new cipher state in the 64-bit version. 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, 7 deposit, 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 costs $3 \cdot 19 \cdot 254 = 14478$ clock cycles (with the operation f_a also costing 19 clock cycles per single bit computation).
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 slided fashion (see Figure 14). 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 Figure 14 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.
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*× 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 is 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).

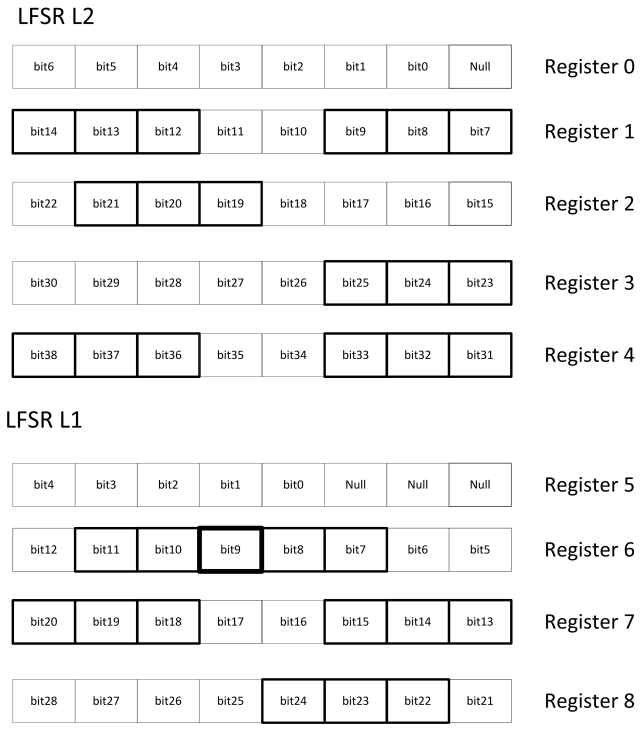

Figure 14: Cipher state of KATAN64, stored in a slided manner, using 9 registers. The bit triads required for computing the new cipher state are highlighted in bold.

```
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
```

Figure 15: 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 (Figure 14 register i corresponds to s_i), and variable t_j denotes temporary values.

3.4 Performance

In this section we present benchmarking results of our implementation. Specifically, we give the attained latency, throughput and size of the implementation and compare to previous implementations. We show that for the price of increased memory consumption, we can increase throughput and decrease latency. The results were obtained on the Atmel Visual studio simulator that allows us to examine the behavior and performance of AVR ATtiny45.

Component cost

The component cost of the KATAN cipher can be viewed in the following array (Table 8). Components include the two non-linear functions and rotation (clocking) of the LFSRs. The key schedule is not computed on the fly, but it is precomputed, stored in SRAM and fetched when needed via 8-bit memory lookups. We note that the non-traditional representation we use has an impact on the rotation layer (increasing it to 39 clock cycles instead of 24 clock cycles, achievable in a standard representation). Still, the non-standard representation allows us to reduce the cost of f_a, f_b by a factor of 3 (38 cc in non-standard for both *vs.* 114 in non-standard for both). Thus, we can achieve increased throughput and latency.

Component	**Cost per round**	**Total cost**
f_a	19	4826
f_b	19	4826
Rotation	39	9906
Key update	precomputed	fetched in 128
Total (254 rounds)	95	23671

Table 8: Component cost (in clock cycles) of the proposed KATAN64 implementation.

Throughput, latency and size

The only known implementation of KATAN64 in AVR architecture is presented by Eisenbarth *et al.* [23] 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 in software; with the penalty of a single extra register, we manage to increase the throughput of the non-linear layer threefold. With regards to key precomputation, the penalty is more steep; we need 74 bytes of SRAM storage to reduce the key update to 64 SRAM memory fetches. The results can also bee seen in Table 9. In addition, we note that the KATAN64 implementation is not bitsliced, thus, any improvement in throughput, automatically translates to lower latency. To conclude, we also include the size metrics in Table 9. Eisenbarth *et al.* result in an implementation that requires 338 flash memory bytes and 18 bytes of SRAM for both encryption and decryption. This work requires 380 flash memory bytes and 96 SRAM bytes (32 for IR and 64 for precomputed key) for encryption only.

KATAN64 implementation	**Throughput (cc)**	**Size (bytes)**
Eisenbarth *et al.* [23]	72063	338 flash, 18 SRAM
This work	23671	380 flash, 96 SRAM

Table 9: Throughput of KATAN64 cipher implementations for AVR architecture, *i.e.*, clock cycles required for a single encryption round. Size of the second row applies to encryption only.

4 PRINCE cipher

In this section, we present part of on-going research pertaining to the PRINCE cipher [12]. Specifically, we solely focus on the substitution and nibble[14] permutation operations of the PRINCE cipher. We suggest a novel idea, namely a variation of the bitslicing technique, in order to efficiently compute these operations. We commence with an overview of the required PRINCE cipher operations (Section 4.1), we motivate our choice in section 4.2 and suggest the new technique in section 4.3.

4.1 Introduction

PRINCE is a 64-bit block cipher with a 128-bit keys, based on the F–X construction[15] [12, 40]. 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:

$$k_0||k_1 \rightarrow k_0||k_0'||k_1 = (k_0||k_0 >>> 1) \oplus (k_0 >> 63||k_1) \tag{7}$$

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. Figure 16 demonstrates the usages of the whitening keys, while Figure 17 shows the 12 rounds of encryption. We also note that PRINCE was crafted with involution[16] in mind, such that reversing the procedure (*i.e.* decrypting) did not cost additional hardware gates.

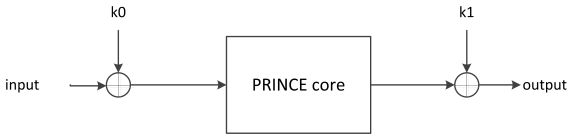

Figure 16: PRINCE core and whitening keys.

[14] 1 nibble = 4 bits.
[15] Francois-Xavier standaert.
[16] An involutary function, is a function f that is its own inverse.

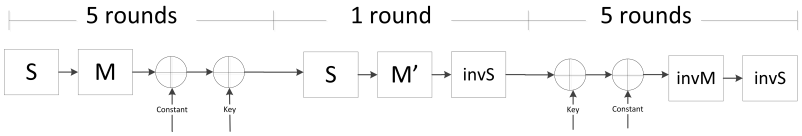

Figure 17: The 12 rounds of the PRINCE cipher. k_1 denotes the core cipher key, RC_is are constants, S the S-box and M the diffusion layer.

Cipher components

`S-layer.` The cipher uses a 4-bit S-box, the action of which is given by the following table (Table 10). We also note that the designers of the PRINCE cipher provide us with alternative S-boxes (acquired via affine transformations) that have the same security properties but may be less or more efficient in hardware implementations.

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 10: The S-Box of the PRINCE cipher.

`Diffusion layer`. The diffusion layer, denoted by M, is constructed by two parts: SR and M'. The SR operation behaves similarly to the AES ShiftRows and permutes the 16 nibbles in the a specific pattern (see Figure 18). The M' part is a matrix multiplication; the current cipher state is multiplied by a 64×64 bit matrix which was constructed from 4-bit-based components. The M' operation is an involution and can be viewed in Figure 19.

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

Figure 18: Nibble permutation of the PRINCE cipher in the SR operation (from old nibble position to new nibble position).

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

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

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

Figure 19: The M' operation, analyzed from top to bottom.

4.2 Implementation motivation

For the PRINCE cipher, we have identified a single motivation point: the structural similarities between the the PRESENT S and P-layer and the PRINCE S and R operations. In fact we attempt to exploit the fact that PRINCE R operation is a permutation with fewer 'degrees of freedom' compared to the PRESENT P-layer.
We show in the following section that relaxing the permutation requirements (*i.e.* PRINCE operation R being simpler than the PRESENT P-layer) can offer us new implementation choices, leading to better overall performance.

4.3 Nibble-slicing the PRINCE cipher

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; it consists of a 4-bit S-box that gets a plaintext nibble (half byte) and returns the substituted nibble. As in PRESENT, we could opt to either implement it with lookup tables or via a multi-output boolean function.
Moving to the linear diffusion part, we observe the following difference between the PRESENT P-layer and the PRINCE R operation (note that we do not discuss the matrix multiplication used for diffusion in PRINCE, named operation M. Efficient implementation of M is intended as future work). The PRESENT P-layer consists of permutations on a bit level, while PRINCE permutations always address 4-bit chunks or nibbles. Recalling our discussion in section 2.5, we note that although implementing the PRESENT

P-layer with bitslicing would cost zero cycles in theory, we cannot achieve this in reality. Storing and maintaining 64 state 'bits' in bitsliced form (namely, maintaining a 64-byte state) was not possible without using the SRAM and as a result, costly memory fetch/store operations.
While there is no alternative way to implement bitslicing if the permutation layer operates on single bits, we observed that PRINCE permutations are always carried on nibbles. Thus, 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 (both in PRINCE and PRESENT).
Nibble-slicing uses the following representation: every 8-bit register is split in two parts (high and low, 4 bits each) and we use a total of 16 registers. The whole representation consists of 128 bits, *i.e.* two separate cipher states (we refer to them as block 1 and block 2 – see Figure 20). 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 [39] or Groestl [51], albeit our technique applies on half-bytes.

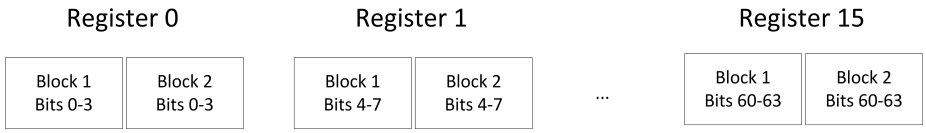

Figure 20: Nibble-sliced representation of the PRINCE cipher.

At this point we observe that 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:

- First, 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.

- Second, although it is still possible, we no longer have to compute the

S-box via a boolean function and we can use the more efficient lookup tables.

Although we demonstrated in chapter 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 [23] and Papagiannopoulos and Verstegen [50], we use a variation of the squared lookup table for S-box computation. During the lookup, each one of 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.
The nibble-sliced representation is also constructed to simplify nibble permutations. For instance, assume that nibble 1 is permuted to the position of nibble 5 during the R operation. To perform this, we no longer have to compute shifts, bit extractions/deposits or such operations; we need only to move the contents of the register that contains nibble 1 to the register containing nibble 5. As a result, both the high and the low nibble are permuted within a single clock cycle (1 `mov` instruction), *i.e.* we achieve single nibble permutation on two cipher blocks within a single clock cycle.
In fact, we can further decrease it (to zero cycles) if we observe that the permutation operation R can be combined with the substitution operation before. During substitution, we use the instruction `lpm destination, address in Z` that substitutes the values on `ZL` according to the S-box, via the lookup. Subsequently, the result is stored in the destination register. Assuming that value `0x3` is stored in the lower half of the register pertaining to nibble position 1 and the value `0x6` is stored in the higher half of the register also pertaining to nibble position 1, we do the following: we substitute the value 0x63 with `0x92` (according to the S-box) and the `lpm` instruction stores it in the register pertaining to nibble position 5. A side-effect is that this procedure may rewrite values while permuting (*e.g.* if the nibble position 5 is not substituted yet). To tackle this, we use a similar technique as in section 2.5.3: we double the register space and use 32 registers for the operation, while alternating between the two register parts (part 1 being registers `r15`...`r30` and part 2 being registers `r0`...`r15`).
Using the afore mentioned representation and techniques, we manage to perform the substitution and nibble-permutation of two PRINCE cipher blocks (due to the nibble-slice factor 2) in 64 clock cycles. Measurements again use the Atmel Visual Studio Simulator. The operations *S,SR* cost

only $32 = 64/2$ clock cycles. Future work intends on including an efficient implementation of the matrix multiplication M used in the PRINCE cipher.

5 Conclusions

Having reached the conclusion section, we aim at providing a knowledge recap and pointers towards interesting future work topics.
This work has managed to achieve increased throughput in both PRESENT and KATAN64 ciphers for AVR ATtiny. The bisliced PRESENT implementation manages a full 64-bit block encryption in 2967 clock cycles, albeit with increased memory requirements (3816 flash memory bytes and 256 SRAM bytes for encryption). The implementation achieves the highest known throughput, outperforming the fastest bitsliced implementation by a factor of *2.1* and the fastest non-bitsliced implementation by a factor of *2.9*. The improved KATAN64 implementation outperforms the only known KATAN64 AVR implementation by a factor of *3* and it executes in 23671 clock cycles, while requiring 380 flash memory bytes and 96 SRAM bytes for encryption. Last, we manage to perform the first two core operation of the PRINCE cipher (substitution and nibble permutation) in 32 clock cycles per round with fairly small memory consumption (256-byte lookup table in flash memory).
With respect to future work, it is useful to examine more lightweight ciphers under the bitslicing prism and construct additional high-throughput implementations; an natural extension is PRINCE. An additional target for future work revolves around examining computationally efficient side channel countermeasures on the AVR ATtiny architecture. We have observed that several different side-properties arise from the various representations or techniques used, thus we can research into how different implementation methods interface with the various countermeasures, *e.g.* if countermeasures become more or less computationally intensive. Last, this line of work can also be attempted in different platforms, such as AVR ATmega, ARM, *etc.* in order to be able to 'power' a large range of devices.

References

[1] Circuit minimization results obtained at Yale University. http://cs-www.cs.yale.edu/homes/peralta/CircuitStuff/CMT.html. URL retrieved: 15-11-2013.

[2] ISO/IEC 29192-2:2011, Information technology - Security techniques - Lightweight cryptography - Part 2: Block ciphers. 2011.

[3] Farzaneh Abed, Christian Forler, Eik List, Stefan Lucks, and Jakob Wenzel. Biclique cryptanalysis of the PRESENT, LED and KLEIN. *IACR Cryptology ePrint Archive*, 2012:591, 2012. URL retrieved: 18-11-2013. http://eprint.iacr.org/2012/591.pdf.

[4] Ross Anderson, Eli Biham, and Lars Knudsen. Serpent: A Proposal for the Advanced Encryption Standard. *NIST AES Proposal*, 1998. URL retrieved: 18-11-2013. http://www.cl.cam.ac.uk/~rja14/Papers/serpent.pdf.

[5] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Mara Naya-Plasencia. Quark: A lightweight hash. *Journal of Cryptology*, 26(2):313–339, 2013. URL retrieved: 18-11-2013. https://131002.net/quark/quark_full.pdf.

[6] Steve Babbage and Matthew Dodd. The stream cipher MICKEY 2.0, ECRYPT stream cipher. 2006. URL retrieved: 18-11-2013. http://www.ecrypt.eu.org/stream/p3ciphers/mickey/mickey_p3.pdf.

[7] Eli Biham. A Fast New DES Implementation in Software. *Technion*, Technical Report CS0891, 1997. URL retrieved: 18-11-2013. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.52.5429&rep=rep1&type=pdf.

[8] Matt Bishop. An application of a Fast Data Encryption Standard Implementation. *Computing Systems*, (3):221–254, 1988. URL retrieved: 18-11-2013. http://www.cs.dartmouth.edu/reports/TR88-138.pdf.

[9] Andrey Bogdanov, Miroslav Knezevic, Gregor Leander, Deniz Toz, Kerem Varici, and Ingrid Verbauwhede. SPONGENT: A lightweight hash function. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 312–325. Springer,

2011. URL retrieved: 18-11-2013. http://homes.esat.kuleuven.be/~abogdano/papers/spongent_ches11.pdf.

[10] Andrey Bogdanov, Lars Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew Robshaw, Yannick Seurin, and Charlotte Vikkelsoe. PRESENT: An ultra-lightweight block cipher. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems – CHES 2007*, volume 4727 of *Lecture Notes in Computer Science*, pages 450–466. Springer, 2007. URL retrieved: 18-11-2013. http://homes.esat.kuleuven.be/~abogdano/papers/present_ches07.pdf.

[11] Andrey Bogdanov, Gregor Leander, Christof Paar, Axel Poschmann, Matt JB Robshaw, and Yannick Seurin. Hash Functions and RFID Tags: Mind the Gap. In Elisabeth Oswald and Pankaj Rohatgi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2008*, volume 5154, pages 283–299. Springer, 2008. URL retrieved: 18-11-2013. http://www.iacr.org/archive/ches2008/51540279/51540279.pdf.

[12] Julia Borghoff, Anne Canteaut, Tim Guneysu, Elif Bilge Kavun, Miroslav Knezevic, Lars R. Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, Peter Rombouts, Soren S. Thomsen, and Tolga Yalcin. PRINCE – a low-latency block cipher for pervasive computing applications. *IACR Cryptology ePrint Archive*, 2012:529, 2012. URL retrieved: 18-11-2013. http://eprint.iacr.org/2012/529.pdf.

[13] Joan Boyar and Peralta. A new combinational logic minimization technique with applications to cryptology. In Paola Festa, editor, *Experimental Algorithms*, volume 6049 of *Lecture Notes in Computer Science*, pages 178–189. Springer, 2010. URL retrieved: 18-11-2013. http://eprint.iacr.org/2009/191.pdf.

[14] Joan Boyar and Peralta. A small depth-16 circuit for the AES S-box. In Furnell Dimitris Gritzalis, Steven and Marianthi Theoharidou, editors, *Information Security and Privacy Research*, volume 376 of *IFIP Advances in Information and Communication Technology*, pages 287–298. Springer, 2011. URL retrieved: 18-11-2013. http://eprint.iacr.org/2011/332.pdf.

[15] Christophe De Canniere, Orr Dunkelman, and Miroslav Knezevic. Katan and ktantan - a family of small and efficient hardware-oriented

block ciphers. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems – CHES 2009*, volume 5747 of *Lecture Notes in Computer Science*, pages 272–288. Springer, 2009. URL retrieved: 18-11-2013. `http://www.cs.technion.ac.il/~orrd/KATAN/CHES2009.pdf`.

[16] Christophe De Canniere and Bart Preneel. Trivium. In Matthew J. B. Robshaw and Olivier Billet, editors, *New Stream Cipher Designs*, volume 4986 of *Lecture Notes in Computer Science*, pages 244–266. Springer, 2008. URL retrieved: 18-11-2013. `http://www.ecrypt.eu.org/stream/p3ciphers/trivium/trivium_p3.pdf`.

[17] Baudoin Collard and F-X Standaert. A Statistical Saturation Attack Against the Block Cipher PRESENT. In Marc Fischlin, editor, *Topics in Cryptology – CT-RSA 2009*, volume 5473 of *Lecture Notes in Computer Science*, pages 195–210. Springer, 2009.

[18] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (2nd Ed.)*. The MIT Press, 2001.

[19] Nicolas Courtois, Daniel Hulme, and Theodosis Mourouzis. Solving circuit optimisation problems in cryptography and cryptanalysis. *IACR Cryptology ePrint Archive*, 2011:475, 2011. URL retrived: 18-11-2013. `http://www.ima.org.uk/_db/_documents/Courtois.pdf`.

[20] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES – The Advanced Encryption Standard*. Springer, 2002.

[21] Atmel datasheet. Atmel 8-bit AVR insruction set. URL retrieved: 18-11-2013, `http://www.atmel.com/images/doc0856.pdf`.

[22] Atmel datasheet. Atmel 8-bit AVR microcontroller datasheet. URL retrieved: 18-11-2013, `http://tinyurl.com/klld65e`.

[23] Thomas Eisenbarth, Zheng Gong, Tim Gneysu, Stefan Heyse, Sebastiaan Indesteege, Stphanie Kerckhof, Franois Koeune, Tomislav Nad, Thomas Plos, Francesco Regazzoni, Franois-Xavier Standaert, and Loc van Oldeneel tot Oldenzeel. Compact implementation and performance evaluation of block ciphers in attiny devices. In Aikaterini Mitrokotsa and Serge Vaudenay, editors, *Progress in Cryptology – AFRICACRYPT 2012*, volume 7374 of *Lecture Notes in Com-*

puter Science, pages 172–187. Springer, 2012. URL retrieved:. http://perso.uclouvain.be/fstandae/PUBLIS/108.pdf.

[24] Thomas Eisenbarth, Timo Kasper, Christof Paar, and Sebastiaan Indesteege. *Encyclopedia of Cryptography and Security (2nd Ed.)*. Springer, 2011.

[25] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, September 1972.

[26] Carsten Fuhs and Peter Schneider-Kamp. Synthesizing shortest linear straight-line programs over GF(2) using SAT. In Ofer Strichman and Stefan Szeider, editors, *Theory and Applications of Satisfiability Testing – SAT 2010*, volume 6175 of *Lecture Notes in Computer Science*, pages 71–84, 2010. URL retrieved: 18-11-2013. http://tinyurl.com/odfxblg.

[27] Brian R. Gladman. Software for efficient boolean function decompositions for the eight Serpent S-boxes and their inverses. URL retrieved: 18-11-2013 http://gladman.plushost.co.uk/oldsite/cryptography_technology/serpent/index.php.

[28] Mahadevan Gomathisankaran and Ruby B Lee. Maya: A novel block encryption function. 2009. URL retrieved: 18-11-2013. http://palms.princeton.edu/system/files/maya.pdf.

[29] Zheng Gong, Pieter H. Hartel, Svetla Nikova, and Bo Zhu. Towards secure and practical MACs for body sensor networks. In Bimal K. Roy and Nicolas Sendrier, editors, *Progress in Cryptology – INDOCRYPT 2009*, volume 5922 of *Lecture Notes in Computer Science*, pages 182–198. Springer, 2009. URL retrieved: 18-11-2013. http://doc.utwente.nl/67473/3/Towards_Secure_and_Practical_MACs_for_Body_Sensor_Networks.pdf.

[30] Zheng Gong, Svetla Nikova, and Yee Wei Law. Klein: A new family of lightweight block ciphers. In Ari Juels and Christof Paar, editors, *RFID, Security and Privacy*, volume 7055 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2011. http://doc.utwente.nl/73129/1/The_KLEIN_Block_Cipher.pdf.

[31] Zheng Gong and Bo Zhu. Software Implementation of Block Cipher PRESENT for 8-Bit Platforms, 2013. URL retrieved: 09-11-2013.

http://cis.sjtu.edu.cn/index.php/Software_Implementation_of_Block_Cipher_PRESENT_for_8-Bit_Platforms.

[32] Jian Guo, Thomas Peyrin, and Axel Poschmann. The PHOTON family of lightweight hash functions. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 222–239. Springer, 2011. URL retrieved: 18-11-2013. http://www.ecrypt.eu.org/hash2011/proceedings/hash2011_04.pdf.

[33] Jian Guo, Thomas Peyrin, Axel Poschmann, and Matthew J. B. Robshaw. The LED block cipher. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 326–341. Springer, 2011. URL retrieved: 11-8-2013. http://eprint.iacr.org/2012/600.pdf.

[34] Martin Hell, Thomas Johansson, and Willi Meier. Grain: a stream cipher for constrained environments. In *International Journal of Wireless and Mobile Computing*, volume 2, pages 86–93, 2007. URL retrieved: 18-11-2013. http://www.ecrypt.eu.org/stream/ciphers/grain/grain.pdf.

[35] Shoichi Hirose. Provably Secure Double-Block-Length Hash Functions in a Black-Box Model. In Choon sik Park and Seongtaek Chee, editors, *Information Security and Cryptology – ICISC 2004*, volume 3506 of *Lecture Notes in Computer Science*, pages 330–342. Springer, 2005. URL retrieved: 18-11-2013. http://repo.flib.u-fukui.ac.jp/dspace/bitstream/10098/2310/1/p075.pdf.

[36] Shoichi Hirose. Some Plausible Constructions of Double-Block-Length Hash Functions. In Matthew Robshaw, editor, *Fast Software Encryption*, volume 4047 of *Lecture Notes in Computer Science*, pages 210–225. Springer, 2006. URL retrieved: 18-11-2013. http://www.iacr.org/archive/fse2006/40470213/40470213.pdf.

[37] Deukjo Hong, Jaechul Sung, Seokhie Hong, Jongin Lim, Sangjin Lee, Bonseok Koo, Changhoon Lee, Donghoon Chang, Jesang Lee, Kitae Jeong, Hyun Kim, Jongsung Kim, and Seongtaek Chee. Hight: A new block cipher suitable for low-resource device. In Louis Goubin and Mitsuru Matsui, editors, *Cryptographic Hardware and Embedded Systems – CHES 2006*, volume 4249 of *Lecture Notes in Com-*

puter Science, pages 46–59. Springer, 2006. URL retrieved: 18-11-2013. http://www.iacr.org/cryptodb/archive/2006/CHES/04/04.pdf.

[38] Michael Hutter and Peter Schwabe. NaCl on 8-bit AVR microcontrollers. In Abderrahmane Nitaj Amr Youssef and Aboul Ella Hassanien, editors, *Progress in Cryptology – AFRICACRYPT 2013*, volume 7918 of *Lecture Notes in Computer Science*, pages 156–172. Springer, 2013. URL retrieved: 18-11-2013. http://cryptojedi.org/papers/avrnacl-20130514.pdf.

[39] Emilia Kasper and Peter Schwabe. Faster and timing-attack resistant AES-GCM. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems – CHES 2009*, volume 5747 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2009. URL retrieved: 18-11-2013. http://eprint.iacr.org/2009/129.

[40] Joe Kilian and Phillip Rogaway. How to protect DES against exhaustive key search. In Neal Koblitz, editor, *Advances in Cryptology – CRYPTO 96*, volume 1109 of *Lecture Notes in Computer Science*, pages 252–267. Springer, 1996. URL retrieved: 18-11-2013. http://www.cs.ucdavis.edu/~rogaway/papers/desx.pdf.

[41] Robert Konighofer. A fast and cache-timing resistant implementation of the AES. In Tal Malkin, editor, *Topics in Cryptology – CT-RSA 2008*, volume 4964 of *Lecture Notes in Computer Science*, pages 187–202. Springer, 2008. URL retrieved: 18-11-2013. https://online.tugraz.at/tug_online/voe_main2.getvolltext?pCurrPk=47852.

[42] Gregor Leander. Small scale variants of the block cipher present. *IACR Cryptology ePrint Archive*, 2010:143, 2010. URL retrieved: 18-11-2013. http://eprint.iacr.org/2010/143.pdf.

[43] Ruby Lee. Accelerating multimedia with enhanced microprocessors. In *IEEE Micro*, volume 15, pages 22–32, 1995. URL retrieved: 18-11-2013. http://www.princeton.edu/~rblee/HPpapers/accelMultimediawEnhancedMicroproc.pdf.

[44] Ruby B. Lee. Subword permutation instructions for two-dimensional multimedia processing in MicroSIMD architectures. In *ASAP*, pages 3–14. IEEE Computer Society, 2000. URL retrieved: 18-11-2013. http://www.princeton.edu/~rblee/PUpapers/lee_asap00.pdf.

[45] Ruby B. Lee, Zhijie Shi, and Xiao Yang. Cryptography efficient permutation instructions for fast software. In *IEEE Micro*, volume 21, pages 56–69, 2001. URL retrieved: 18-11-2013. `http://palms.ee.princeton.edu/PALMSopen/lee01efficient.pdf`.

[46] Chae Hoon Lim and Tymur Korkishko. mCrypton - a lightweight block cipher for security of low-cost RFID tags and sensors. In JooSeok Song, Taekyoung Kwon, and Moti Yung, editors, *Information Security Applications*, volume 3786 of *Lecture Notes in Computer Science*, pages 243–258. Springer, 2005.

[47] Daniel Hulme Guangyan Song Martin Albrecht, Nicolas T. Courtois. Bit-slice implementation of PRESENT in pure standard C. URL retrieved: 18-11-2013. `https://bitbucket.org/malb/research-snippets/src`.

[48] Jorge Nakahara Jr, Pouyan Sepehrdad, Bingsheng Zhang, and Meiqin Wang. Linear (hull) and Algebraic Cryptanalysis of the Block Cipher PRESENT. In Atsuko Miyaji Juan A. Garay and Akira Otsuka, editors, *Cryptology and Network Security*, volume 5888 of *Lecture Notes in Computer Science*, pages 58–75. Springer, 2009. URL retrieved: 18-11-2013. `http://www.ioc.ee/~tarmo/tday-meintack/zhang-slides.pdf`.

[49] Kostas Papagiannopoulos. Speed-optimized implementation of PRESENT in AVR assembly. 2013. URL retrieved: 18-11-2013. `https://github.com/kostaspap88/PRESENT_speed_implementation/`.

[50] Kostas Papagiannopoulos and Aram Verstegen. Speed and size-optimized implementations of the PRESENT cipher for Tiny AVR devices. In Michael Hutter and Jorn-Marc Schmidt, editors, *Radio Frequency Identification*, Lecture Notes in Computer Science, pages 161–175, 2013.

[51] Krystian Matusiewicz Florian Mendel Christian Rechberger Martin Schlaffer Praveen Gauravaram, Lars R. Knudsen and Soren S. Thomsen.

[52] The GNU project. AVR-Crypto-Lib. 2013. URL retrieved: 06-05-2013. `http://avrcryptolib.das-labor.org/`.

[53] Pablo Rauzy, Sylvain Guilley, and Zakaria Najm. Formally proved security of assembly code against leakage. *IACR Cryptology ePrint*

Archive, page 554, 2013. URL retrieved: 18-11-2013. http://eprint.iacr.org/2013/554.pdf.

[54] Chester Rebeiro, A. David Selvakumar, and A. S. L. Devi. Bitslice implementation of AES. In David Pointcheval, Yi Mu, and Kefei Chen, editors, *Cryptology and Network Security*, volume 4301 of *Lecture Notes in Computer Science*, pages 203–212. Springer, 2006.

[55] Zhijie Shi and Ruby B. Lee. Bit permutation instructions for accelerating software cryptography. In *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pages 138–148, 2000. http://www.princeton.edu/~rblee/PUpapers/shi_asap00.pdf.

[56] Kyoji Shibutani, Takanori Isobe, Harunaga Hiwatari, Atsushi Mitsuda, Toru Akishita, and Taizo Shirai. Piccolo: An ultra-lightweight blockcipher. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 342–357. Springer, 2011.

[57] Taizo Shirai, Kyoji Shibutani, Toru Akishita, Shiho Moriai, and Tetsu Iwata. The 128-bit blockcipher CLEFIA (extended abstract). In Alex Biryukov, editor, *FSE*, volume 4593 of *Lecture Notes in Computer Science*, pages 181–195. Springer, 2007. URL retrieved: 18-11-2013. http://www.iacr.org/archive/fse2007/45930182/45930182.pdf.