AVRprince - An Efficient Implementation of PRINCE for 8-bit Microprocessors

Aria Shahverdi, Cong Chen, and Thomas Eisenbarth

Worcester Polytechnic Institute, Worcester, MA, USA
{ashahverdi,cchen3,teisenbarth}@wpi.edu

Abstract. PRINCE is a lightweight block cipher that has been optimized for low latency and a small hardware footprint. This work presents two 8-bit oriented microcontroller implementations of PRINCE. The implementations are optimized to achieve good throughput at moderate code size. One implementation applies the T-table style to PRINCE. The other implementation is optimized to process two nibbles in parallel. This parallel processing increases throughput, but requires a restructuring of the state for efficient computation of the MixColumns, as well as an increased code size for the two double S-boxes. Due to the structure of PRINCE, the decryption comes at a negligible code overhead. Implementation results are compared to other state-of-the-art implementations on the same platform.

1 Motivation

Implementation of lightweight ciphers is of particular interest in deeply embedded applications, where cost constraints are usually a dominating factor. In these scenarios lightweight block ciphers such as PRINCE can be an advantageous choice, as they might offer significant advantages in power or energy consumption and area or code size, and thus system cost. Having efficient software implementations of hardware-oriented ciphers such as PRINCE is of relevance: In many cases, one side of the communication is implemented in hardware, but the other side might still be implemented in software, to maintain higher flexibility. Examples include RFID tags and readers: the tag is almost always a hardware implementation while readers are usually implemented in software.

In this work two efficient implementations of PRINCE for low-level microprocessors—the AVR family—are presented. Performance of the implementation is compared to other implementations for the same platform, as presented in [4] and in [5]. After a description of the PRINCE cipher and the ciphers used for performance comparison in Section 2, the two implementations and applied optimizations are described in Sections 3 and. 4 The performance of the implementation is discussed in Section 5, followed by a conclusion.

2 The PRINCE Block Cipher

Several lightweight block ciphers have been proposed with the goal of being a low footprint in hardware or good performance at small memory footprint in software. Examples include ciphers like Present, TEA, HIGHT, etc. An overview of implementation properties can be found in [4]. Among these, PRINCE is a hardware-oriented lightweight block cipher that has been optimized for low latency and a small area footprint [2]. It features a 64-bit block size, 128-bit key size. Note that the claimed security level depends on the number of observable plaintext/ciphertext pairs and is given as $127 - n$ bits, where 2^n is the number of attacker-observable plaintext/ciphertext pairs. PRINCE implements a substitution-permutation network which iterates for 12 rounds. The round function is AES-like and operates on a 4-by-4 array of nibbles, featuring 4-bit S–box, as well as ShiftRows and MixColumns operations. The round key remains constant, but is augmented with a 64-bit round constant to ensure variation between rounds. An interesting feature of PRINCE is the inflective property: encryption and decryption only differ in the round key, i.e. decryption can use the same implementation as encryption, only the round key needs to be modified. Figure 1 shows the structure of the PRINCE cipher. To implement PRINCE, the following operations have to be instantiated:

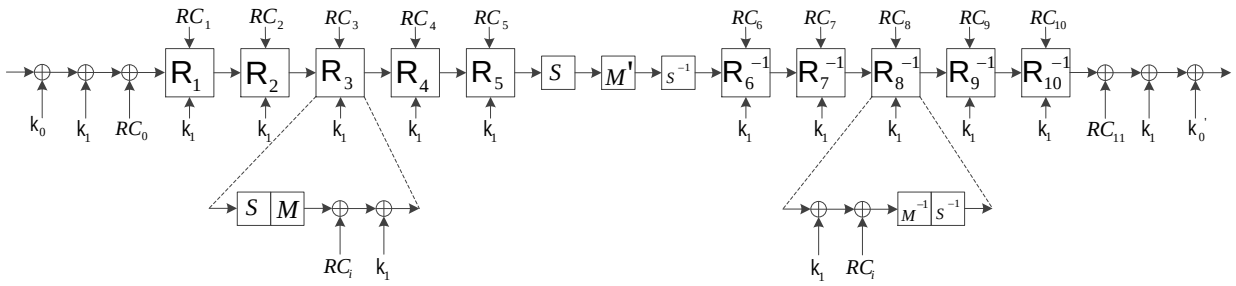

Fig. 1. The PRINCE cipher

Key Schedule The 128-bit key is split into two parts k_0 and k_1. k_0 is used to generate another key $k_0' = (k_0 >>> 1) \oplus (k_0 >> 63)$. The keys k_0 and k_0' are used as pre- and post-whitening keys, i.e. are XOR-added to the state before and after all round functions are performed. The round key k_1 is the same for all rounds and is also XOR-added during the key addition phase.

Round Constant Addition PRINCE defines different round constants RC_i for each round. A noteworthy property of the round constants is that $RC_i \oplus RC_{11-i} = \alpha$ for $0 \leq i \leq 11$, with $\alpha =$ `c0ac29b7c97c50dd`. The round constant addition is a binary addition, just as the round key addition. Both operations can be merged.

S–box The S–box layer uses a mapping of 4-bit to 4-bit, as defined in the following table.

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

Linear Layer The linear layer consists of two parts: a ShiftRows which is similar to the ShiftRows used in AES and simply changes the order of the nibbles. The MixColumns equivalent XOR-adds three input bits to compute one output bit in such a way that the operation is invertible. As can be seen in Table 2, this operation is the most time-consuming operation of the cipher. Our design is optimized in such a way that the time needed for computing this operation is minimized. The details of our method is given in Section 4.

All operations also need an implementation of their inverse, as the last six rounds use the inverse operations. This also means that, unlike most other ciphers, getting rid of either encryption or decryption does not significantly reduce the code size.

3 T-Table implementation of PRINCE

T-table implementations combine different operations within a round into a single table lookup operation. This idea was successfully applied to AES [3] in order to speed up its performance in software implementations. The structure of PRINCE differs from that of AES in two ways: the execution sequence of the round operations is different and—more importantly—the operations and order change for the second half of the cipher. In the following we show that a T-table implementation is still possible.

For the *first five rounds* of PRINCE, we denote $S[\cdot], M[\cdot], SR[\cdot]$ as the S-box lookup, MixColumns and shift row operations, respectively. Then, each round function can be denoted as $s' = k_r \oplus SR[M[S[s]]]$ where s and s' are the block states before and after the round, and k_r is the round key (i.e. $k_r = k_1 \oplus rk_i$). This computation sequence is not suitable for T-table generation. However, the round function can also be written as $s' = SR[SR^{-1}[k_r] \oplus M[S[s]]]$ because ShiftRows is a linear operation. Then, if we take the

$SR[.]$ at the end of each round as the first operation of next round, we have a new round function as $s' = SR^{-1}[k_r] \oplus M[S[SR[s]]]$. Now, we can combine the MixColumns, S-box lookup and ShiftRows as one table lookup as classically done for T-table implementations. Note that the key schedule should include an inverse ShiftRows operation beforehand.

The *involutional middle round* of the cipher $s' = S^{-1}[M[S[s]]]$ can be broke down into two parts. The first part $[M[S[s]]$, together with the ShiftRows from the fifth round, also can be implemented as a T-table lookup using the same table as before. The second part $S^{-1}[.]$ can be merged into the following five rounds, as we show next.

For *the last five rounds*, the round function is originally denoted as $s' = S^{-1}[M[SR^{-1}[k_r \oplus s]]]$ which is not a suitable sequence, either. However, since $S^{-1}[.]$ in the middle of the cipher is taken as the starting point of the following rounds, the round function can be rewritten as $s' = M[SR^{-1}[k_r \oplus S^{-1}[s]]]$. This equation is equivalent to $s' = M[S^{-1}[SR^{-1}[s]]] \oplus M[SR^{-1}[k_r]]$ because ShiftRows is linear. Hence, MixColumns, inverse-S-box lookup and inverse-ShiftRows can be combined as another table lookup operation. For this part, Mixcolumns and inverse-ShiftRows should be performed on the round keys beforehand. Note that an inverse S-box lookup is left over in the end and should be added after the T-table lookup, resulting in an additional small code overhead.

4 Block-Parallel Implementation of PRINCE

Our second implementation aims to maximize throughput while keeping code size moderately low. State and round keys are stored in internal registers to minimize RAM accesses after the initialization. PRINCE is nibble-oriented: in order to achieve better performance, the block-parallel implementation always stores two nibbles in one register and processes them in parallel wherever possible. Storing adjacent nibbles in one register in the natural way results in an inefficiency in mix column layer, since the MixColumns operation differs for the inner and outer columns. To overcome this problem the nibbles are reordered so that nibbles from the inner columns as well as the outer columns are stored together.

The cipher starts with a key whitening in which the state is XOR-added with k_0. Then, as mentioned earlier, the order of the state will change, to be able to process two nibbles at the same time. The order of the round key i.e. k_1 is changed accordingly.

The next part of the cipher is adding the round key alongside adding the round constant. Adding the round key k_1 is done by XOR-adding to the state. Adding the round constant is the same as adding round key but the value of the round constant is stored in a proper format in program memory. The S-boxes are stored as two 256-byte tables which enables the processing of one byte, i.e. two nibbles, at a time.

The next two steps, known as linear layer, are ShiftRows and MixColumns operations. The shift row can be done by masking bits and swapping the registers. The mix column layer is in a way that the first and the last columns as well as the two inner columns are processed in the similar way, as it was mentioned before this is the reason for changing the order of nibbles (Table 1). The reordering enables the parallel processing of two nibbles in the MixColumns operation, effectively doubling the throughput of the operation.

Table 1. Reordering scheme.

Registers	$R0$		$R1$		$R2$		$R3$		$R4$		$R5$		$R6$		$R7$	
Before Reordering	S_0	S_4	S_8	S_{12}	S_1	S_5	S_9	S_{13}	S_2	S_6	S_{10}	S_{14}	S_3	S_7	S_{11}	S_{15}
After Reordering	S_0	S_{12}	S_1	S_{13}	S_2	S_{14}	S_3	S_{15}	S_4	S_8	S_5	S_9	S_6	S_{10}	S_7	S_{11}

5 Implementation Results and Comparison

This section details on the implementation results of PRINCE and compares them to other block cipher implementations on the same platform as well as a comparison between two methods of PRINCE implementation. The code was implemented in Assembly and simulated using AVR Studio 6.1. The target processor is the ATMega 128, as well as the ATtiny 45 used in [4]. For T-table implementation of PRINCE, the code size is 1990 bytes. Both encryption and decryption take 4292 clock cycles. The round keys are precomputed and stored in RAM. The key schedule takes 3833 clock cycles for encryption and 3843 clock cycles for decryption. Each round function takes 349 clock cycles, where each round consists of T-table lookup and key addition which take 283 and 56 clock cycles, respectively. It means that 10 cycles are used for controlling purposes. For block-parallel implementation the code size is 1574 bytes; Encryption of a single block takes 3253 clock cycles, while decryption takes 3293, due to key adaption. The block-parallel implementation has an on-the-fly key schedule, ensuring RAM requirement when compared to the other implementation. Each round function of PRINCE takes 255 clock cycles. Each round of PRINCE consists of S-box, ShiftRows, MixColumns, Round Constant Addition and Round Key Addition. The number of operations done in each round is 244 in total, which means that 11 cycles are used for controlling purposes. Table 2 lists the number of clock cycles, broken down into the different operations, for both implementations.

Table 2. Clock cycle of each operation in one round.

Operation	**Clock Cycle**	**T-table operations**	**Clock Cycle**
$S-box$	40		
$Shift\ Row$	40	$T-table\ lookup$	283
$MixColumns$	111		
$RC_i\ add$	45		
$k_1\ add$	8	$key\ add$	56
Sum	244	**Sum**	339

The PRINCE implementation is compared to AES, HIGHT, IDEA, KATAN, KLEIN, PRESENT and TEA. Reference implementations are taken from [4] and [9] and are freely available online. A more detailed description of the ciphers can be found in [3,8,6,7,1,10].

Table 3 lists the performance results of the presented PRINCE implementation and compares them to related ciphers. Listed code size excludes the testbench, i.e. is for a single run of the cipher. Cycle count and memory consumption are derived using AVR Studio simulations.

We consider three different metrics: code size (in bytes), cycle count in encryption and decryption and a combined metric, namely the code size $\times$ cycle count product, normalized by the block size.

The code size of the presented PRINCE implementation is 1730 bytes, putting it at the upper end of the analyzed lightweight ciphers. This is partially due to the implementation choice of using double S-boxes, which allow for two S-box lookups in a single round, but require 512 bytes of RAM for storing the table. Note that, unlike most of the ciphers compared to, the cost does not decrease if only encryption OR decryption is implemented. In fact, the code for both is nearly identical. For T-table PRINCE, 232 bytes of RAM are used because all the round keys are precomputed and stored in the RAM. Code size results are visualized in Figure 2.

In terms of performance, the block-parallel implementation stays below 3300 clock cycles for both encryption and decryption. This compares favorably to other hardware-oriented block ciphers. The comparably good performance is both due to the low latency cipher design as well as due to the pro-performance optimizations of the implementation. However, well-optimized implementations of e.g. AES are still faster, in spite of the bigger block size. As for the T-table PRINCE, the key scheduling is relatively slow and consumes a lot of memory. This is because inverse-ShiftRows and MixColumns should be performed on the round keys

Table 3. Performance evaluation of our implementation on the AVR microcontroller.

Cipher	Block Size [bits]	Key Size [bits]	Code Size [bytes]	RAM [bytes]	Cycles (enc+key)	Cycles (dec+key)
AES [9]	128	128	1570	-	2739	3579
HIGHT [4]	64	128	402	32	19503	20159
HIGHT [5]	64	128	5672	-	2964	2964
IDEA [4]	64	128	836	232	∼8250	∼22729
IDEA [5]	64	128	596	-	2700	15393
KATAN [4]	64	80	338	18	72063	88525
KLEIN [4]	64	80	1268	18	6095	7658
PRESENT [4]	64	80	1000	18	11342	13599
TEA [4]	64	128	648	24	7408	7539
PRINCE Block-Parallel	64	128	1574	24	3253	3293
PRINCE T-Table	64	128	1990	232	4292+3833	4292+3843

beforehand as discussed in the previous section. Performance results are visualized in Fig. 4 (encryption), Fig. 5 (decryption), and Fig. 3 (combined encryption + decryption).

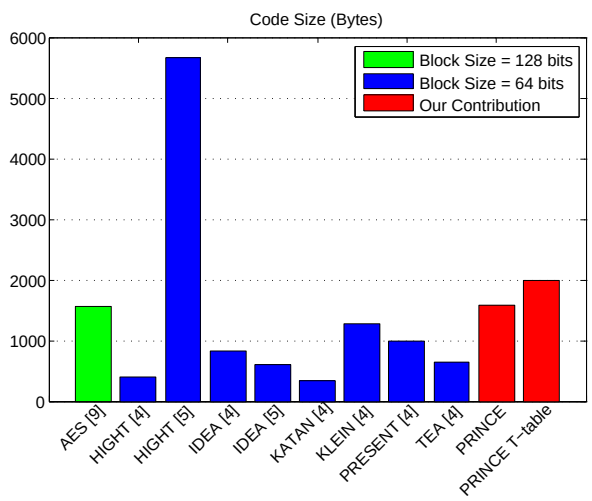

Fig. 2. Code Size: Comparison with Previous Work

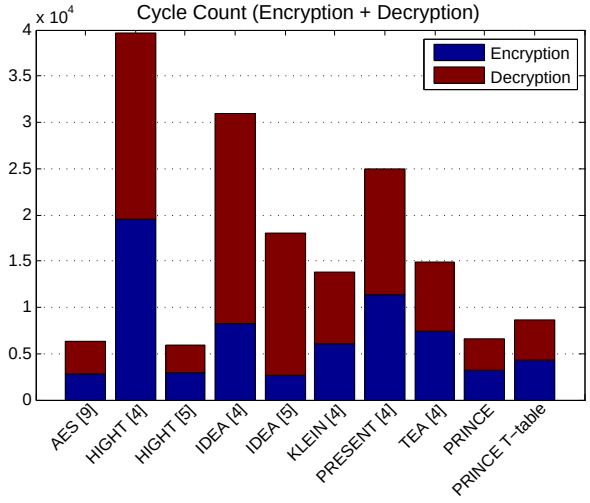

Fig. 3. Performance Evaluation: Cycle Count (Encryption + Decryption)

Following the combined metric used in [4], the PRINCE implementation is, again, somewhere in the middle region, and clearly outperformed by AES. At the same time, it does indeed outperform other hardware-oriented cipher designs such as Present and HIGHT. Results of the combined metric are depicted in Fig. 6 (for encryption) and Fig. 7 (for decryption).

6 Conclusion

A first microcontroller implementation of PRINCE was presented and compared to related implementations on the same platform. With careful optimizations the performance is neither surprisingly good nor bad on the

chosen platform. Since no microcontroller-specific optimizations were used, it is likely that similar implementation techniques will result in comparable relative performance on other low-end 8/16-bit microcontroller platforms.

Code

The source code for both implementations is available. The code for the block-parallel implementation can be found at
`http://ecewp.ece.wpi.edu/wordpress/vernam/projects/source-codes/prince_source_code`.
The T-table version of PRINCE can be found at
`http://ecewp.ece.wpi.edu/wordpress/vernam/projects/source-codes/t-table-prince-source-code/`.

References

1. A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. Robshaw, Y. Seurin, and C. Vikkelsoe. Present: An ultra-lightweight block cipher. In *Cryptographic Hardware and Embedded Systems-CHES 2007*, pages 450–466. Springer, 2007.
2. J. Borghoff, A. Canteaut, T. Güneysu, E. B. Kavun, M. Knezevic, L. R. Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger, P. Rombouts, S. S. Thomsen, and T. Yalcin. Prince – a low-latency block cipher for pervasive computing applications. In *ASIACRYPT*, pages 208–225, 2012.
3. J. Daemen and V. Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer, 2002.
4. T. Eisenbarth, Z. Gong, T. Güneysu, S. Heyse, S. Indesteege, S. Kerckhof, F. Koeune, T. Nad, T. Plos, F. Regazzoni, F.-X. Standaert, and L. Oldeneel tot Oldenzeel. Compact implementation and performance evaluation of block ciphers in attiny devices. In A. Mitrokotsa and S. Vaudenay, editors, *Progress in Cryptology - AFRICACRYPT 2012*, volume 7374 of *Lecture Notes in Computer Science*, pages 172–187. Springer Berlin Heidelberg, 2012.
5. T. Eisenbarth, S. Kumar, C. Paar, A. Poschmann, and L. Uhsadel. A survey of lightweight-cryptography implementations. *Design & Test of Computers, IEEE*, 24(6):522–533, 2007.
6. D. Hong, J. Sung, S. Hong, J. Lim, S. Lee, B.-S. Koo, C. Lee, D. Chang, J. Lee, K. Jeong, et al. Hight: A new block cipher suitable for low-resource device. In *Cryptographic Hardware and Embedded Systems-CHES 2006*, pages 46–59. Springer, 2006.
7. X. Lai and J. L. Massey. A proposal for a new block encryption standard. In *Advances in CryptologyEUROCRYPT90*, pages 389–404. Springer, 2006.
8. G. Leander, C. Paar, A. Poschmann, and K. Schramm. New lightweight des variants. In *Fast Software Encryption*, pages 196–210. Springer, 2007.
9. B. Poettering. Rijndaelfurious aes-128 implementation for avr devices (2007). `http://point-at-infinity.org/avraes/rijndaelfurious.asm`, 2013.
10. D. J. Wheeler and R. M. Needham. Tea, a tiny encryption algorithm. In *Fast Software Encryption*, pages 363–366. Springer, 1995.

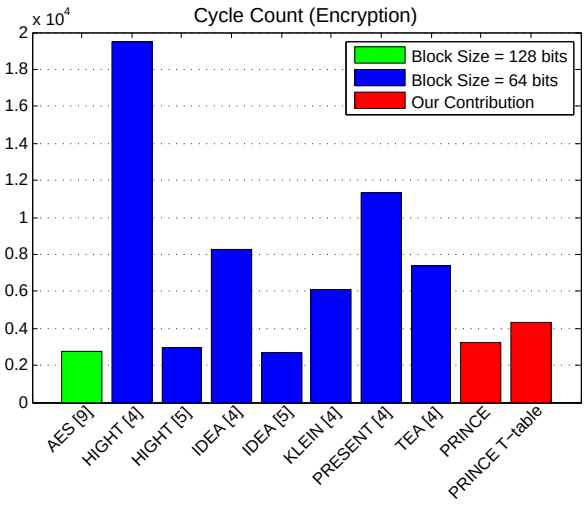

Fig. 4. Performance Evaluation: Cycle Count (Encryption)

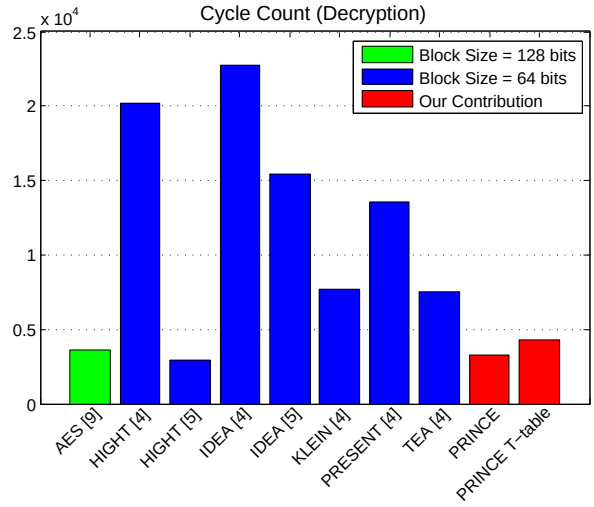

Fig. 5. Performance Evaluation: Cycle Count (Decryption)

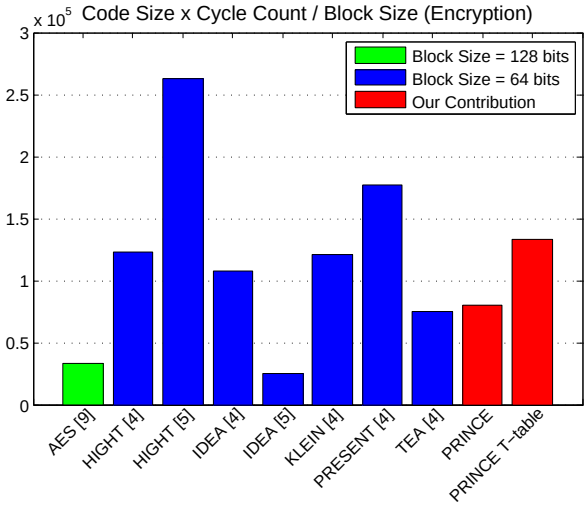

Fig. 6. Performance Evaluation: Combined Metric (Encryption)

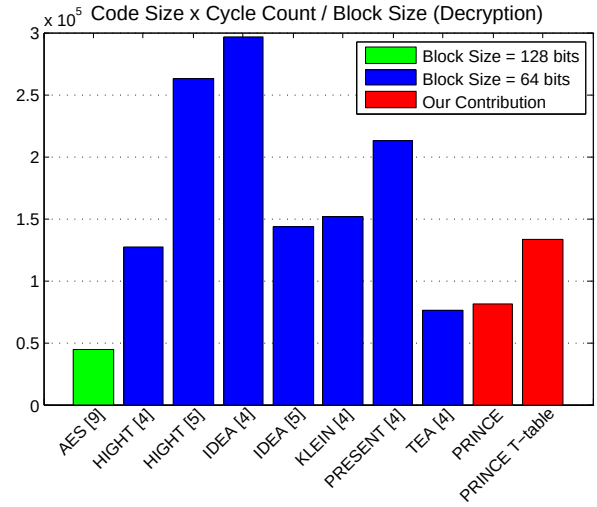

Fig. 7. Performance Evaluation: Combined Metric (Decryption)