Energy Evaluation of Software Implementations of Block Ciphers under Memory Constraints*

Johann Großschädl
jgrosz@iaik.tugraz.at

Stefan Tillich
stillich@iaik.tugraz.at

Christian Rechberger
chrech@iaik.tugraz.at

Michael Hofmann
mhofmann@sbox.tugraz.at

Marcel Medwed
koermy@sbox.tugraz.at

Graz University of Technology
Institute for Applied Information Processing and Communications
Inffeldgasse 16a, A–8010 Graz, Austria

ABSTRACT

Software implementations of modern block ciphers often require large lookup tables along with code size increasing optimizations like loop unrolling to reach peak performance on general-purpose processors. Therefore, block ciphers are difficult to implement efficiently on embedded devices like smart cards or sensor nodes where run-time memory and program ROM are scarce resources. In this paper we analyze and compare the performance, energy consumption, run-time memory requirements, and code size of the five block ciphers RC6, Rijndael, Serpent, Twofish, and XTEA on the StrongARM SA-1100 processor. Most previous evaluations of block ciphers considered performance as the sole metric of interest and did not care about memory requirements or code size. In contrast to previous work, our study of the performance and energy characteristics of block ciphers has been conducted with "lightweight" implementations which restrict the size of lookup tables to 1 kB and also impose constraints on the code size. We found that Rijndael and RC6 can be well optimized for high performance and energy efficiency, while at the same time meeting the demand for low memory (RAM and ROM) footprint. In addition, we discuss the impact of key expansion and modes of operation on the overall performance and energy consumption of each block cipher. Our simulation results show that RC6 is the most energy-efficient block cipher under memory constraints and thus the best choice for resource-restricted devices.

Keywords

Lightweight cryptography, symmetric cipher, energy optimization, memory footprint, code size reduction.

1. INTRODUCTION

Mark Weiser envisioned in the early 1990s a world filled with "ubiquitous" computational resources providing access to information and services anytime, anywhere [19]. Today, 15 years later, part of Weiser's vision has become reality thanks to the proliferation of mobile computing appliances like cell phones and PDAs. Another form of an ubiquitous computing infrastructure are networks of tiny sensor nodes embedded into the environment to perform monitoring and surveillance tasks. Cell phones, PDAs, sensor nodes and the like are typically equipped with wireless networking capabilities, enabling them to communicate with other devices and centralized resources or to connect to the Internet. The networks formed by ubiquitous systems are characterized by a high level of heterogeneity and ad-hoc communication [12]. Middleware plays an essential role in ubiquitous computing because it weaves together a multitude of resources and capabilities (e.g. different types of devices, operating systems, network interfaces, communication protocols) and hides their heterogeneity from the applications [8].

Current research on middleware for ubiquitous computing covers a wide range of topics such as routing, synchronization, quality of service, data aggregation, location awareness (context sensitivity), and resource discovery. In the recent past, however, security issues received particular attention since wireless ad-hoc networks are relatively open and easily accessible, which calls for effective measures to protect the resources and services from unauthorized use. The importance of security becomes evident when considering that, in the near future, every human being will be surrounded by tens or even hundreds of ubiquitous computing devices. For example, wireless sensor nodes are already (or will soon be) deployed in health care for monitoring the blood pressure or heart rate of a patient. Thus, any effort put into making ubiquitous computing systems more secure and reliable as today's PCs is legitimate [16].

Wireless networks transmit data via radio signals, which makes them vulnerable to eavesdropping and unauthorized access. Therefore, the ability to encrypt messages prior to transmission is of fundamental importance for the security of ubiquitous computing systems. Block ciphers and stream ciphers are two examples of symmetric cryptosystems that can be used to encrypt large amounts of data ("bulk encryption"). Roughly speaking, a block cipher takes a fixed-length block of plaintext and a secret key as input and transforms the plaintext into a ciphertext of the same length. Modern block ciphers have a block length of 64 or 128 bits, whereas the key length is typically in the range of between 128 and 256 bits. Research on block ciphers has a long tradition and received particular attention during the competition for the Advanced Encryption Standard (AES), organized by the NIST between 1997 and 1999. Researchers from all

*The research described in this paper has been supported by the Austrian Science Fund (FWF) under grant number P16952-NO4 and by the European Commission under grant number FP6-IST-033563 (project SMEPP).

over the world submitted evaluations of the AES candidates and analyzed both security and performance aspects. However, most performance studies were conducted on high-end processors with superscalar execution pipeline, out-of-order scheduling, and plenty of cache. The findings derived from these evaluations do not allow to draw direct conclusions regarding block cipher performance on embedded processors with a single-issue pipeline and small cache.

In the present paper we analyze performance and energy characteristics of five block ciphers on an embedded RISC processor. These five block ciphers are XTEA and the AES candidates RC6, Rijndael, Serpent, and Twofish. Previous evaluations of block ciphers considered performance as the sole metric of interest and did not care about other aspects like energy consumption, code size, or memory footprint, all of which are important for resource-constrained ubiquitous computing systems such as cell phones, PDAs, and sensor nodes. Contrary to previous work, we conducted our performance and energy analysis with "lightweight" software implementations of block ciphers which restrict the run-time memory usage to 1 kB and also impose limitations on the code size. Optimizing block cipher implementations towards low energy consumption is of paramount importance for virtually any battery-powered device. Also memory usage matters heavily since RAM is a precious resource in sensor nodes. Other devices like PDAs have more RAM, but often relatively small caches, which again calls for low memory footprint. Finally, the code size is important since embedded systems store all program code in on-chip ROM whose size directly determines the cost of a device.

2. SELECTION OF BLOCK CIPHERS AND EVALUATION METHODOLOGY

Block ciphers are cryptographic primitives with a variety of uses. A block cipher defines a key-dependent mapping from input blocks to output blocks. Most block ciphers have a fixed *block size* of either 64 or 128 bits. On the other hand, the *key size* of state-of-the-art block ciphers is often variable, typically ranging from 128 to 256 bits. To allow both a reasonable level of security and a fair comparison, we decided to fix the key size to 128 bits and use 128-bit data blocks for our energy evaluation. These restrictions do not essentially reduce the pool of candidates for our study since almost all modern block ciphers support this setting.

For our energy evaluation we chose the five block ciphers RC6 [13], Rijndael [5], Serpent [1], Twofish [14], and XTEA [10]. The former four were finalists of the AES contest and gained significant practical importance in recent years. We decided to skip Mars, the fifth AES finalist, since it is the "heaviest" block cipher among the AES finalists. Software implementations of Mars require a large lookup table of 2 kB in size, which is clearly disadvantageous for embedded applications where memory resources are at a premium. We selected XTEA as replacement for Mars because it is often advocated as being a lightweight block cipher.

We used ANSI C implementations of the block ciphers to evaluate their performance and energy characteristics. Our C codes are, in part, based on Gladman's highly-optimized implementations of the AES finalists [6]. We simulated all block ciphers with help of Sim-Panalyzer, a cycle-accurate instruction set simulator based on SimpleScalar [18]. Sim-Panalyzer is able to model both the execution time and the energy consumption of software running on a StrongARM SA-1100 processor. ARM-compliant processors, in particular the StrongARM, have a considerable market share in the embedded systems area. Various cell phones and PDAs (e.g. HP Jornada and Compaq iPAQ) are equipped with a StrongARM core or a comparable implementation of the ARM instruction set architecture [2]. In addition, a number of sensor nodes feature ARM processors, most notably the Rockwell WINS node, Intel's iMote, and Sun's SPOT.

Contrary to performance, the energy efficiency of block ciphers has not been widely investigated in the past. To the best of our knowledge, there exist only three publications dealing with energy aspects of software implementations of block ciphers. Law et al. [9] studied the performance and energy characteristics of different block ciphers on a sensor node equipped with a 16-bit MSP430 processor from Texas Instruments. Hager et al. [7] analyzed the energy consumption of the four block ciphers RC2, Blowfish, XTEA, and Rijndael on hand-held computers. Potlapally et al. [11] researched the energy cost of various cryptographic primitives and security protocols on an iPAQ PDA. Both Potlapally et al. and Hager et al. used performance-optimized software implementations for their energy evaluation and did not pay attention to memory requirements.

3. MOTIVATING EXAMPLE

The performance of block ciphers can vary substantially depending on the specific implementation and the properties of the underlying processor. In [17] Tillich et al. examined the influence of the cache size on the performance of four different Rijndael software implementations on a SPARC V8-compliant RISC processor. These four implementations required lookup tables of different size, ranging from 256 byte (only S-box lookup) to 8 kB (T-tables [5]). Figure 1 illustrates the execution time of the implementations as a function of cache size varying from 1 kB to 16 kB (separate instruction and data cache of the same size).

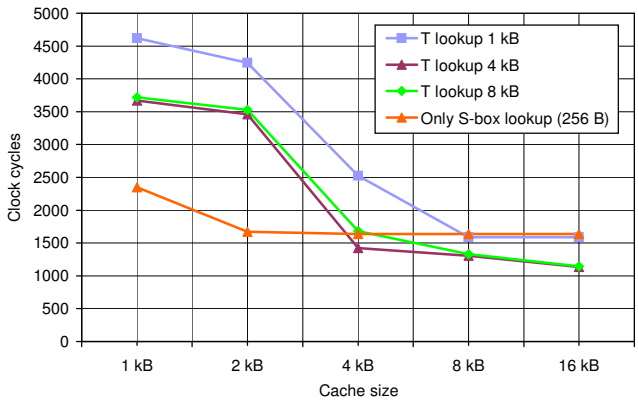

Figure 1: Performance of different Rijndael implementations as a function of the cache size.

The execution time of the implementation with the 256-byte table for S-box lookup is relatively independent of the cache size. It outperforms the other three implementations impressively on small-cache systems. On the other hand, the implementations using large T-tables are fast on systems with large cache, but their performance declines for small cache sizes [17]. In any case, large lookup tables consume a

Operation	RC6	Rijndael	Serpent	Twofish	XTEA
Table lookup	–	8 to 8 bit	–	8 to 8 bit (2×)	–
Logical operation	*xor*, fixed *and*	*xor*	*xor, and, or, not*	*xor*	*xor*, fixed *and*
Shift/rotate	variable rotate	bytewise rotate	fixed shift	fixed rotate	fixed rotate
Integer add/sub	X	–	–	X	X
Integer multiply	X	–	–	–	–
GF(2^8) multiply	–	X	–	X	–

Table 1: Basic operations of the five block ciphers RC6, Rijndael, Serpent, Twofish, and XTEA.

significant amount of program memory (if they are stored as constants) or data memory (when pre-calculated during run-time). For embedded systems, it can not be taken as granted that this price in terms of memory overhead can be paid, even if there is a performance gain. Therefore, the strategy of T-lookup [6], which is a natural implementation choice for Rijndael on desktop computers and servers, can quickly become unattractive for small-memory devices.

4. EXAMINED BLOCK CIPHERS

All block ciphers used in practice have an iterative structure. An input block is encrypted by several, often identical round transformations. The round transformations apply sub-keys that are either computed in advance or in parallel by means of a key-schedule function. Round transformations and the key-schedule function of a block cipher typically consist of a set of simple operations. The block ciphers we examined (RC6, Rijndael, Serpent, Twofish, and XTEA) share a number of fundamental operations. These are table lookups of varying size, bitwise Boolean operations, basic arithmetic operations like integer addition, subtraction, or multiplication, as well as bitwise shift and rotate operations by a fixed or variable number of positions.

Note that the basic operations of each block cipher can be implemented in different ways. However, in this paper we only consider implementation options which are suitable for resource-constrained devices. Some operations allow to trade memory (or code size) for performance/energy. As an example, consider the finite field inversion carried out in the Rijndael round transformation. Even though it is possible to implement it using Boolean functions only, the required execution time—and, hence, the consumed energy—would be very high. Therefore, a lookup into a moderately sized table is usually more suitable, even on memory-restricted devices like sensor nodes. On the other hand, the situation is completely different for the block cipher Serpent. Serpent uses tiny 4-bit by 4-bit S-boxes, which could be easily implemented via lookup tables, even when memory resources are at a premium. However, on 32-bit platforms, it is generally more efficient to perform the S-box operations as a sequence of Boolean functions, following a bit-slice approach [4].

Table 1 summarizes the basic operations of the five block ciphers. Note that we do not distinguish between shift and rotate operations since our target platform (StrongARM) supports both in a similar way.

RC6

RC6 was proposed by Rivest et al. in 1998 [13]. It is an unbalanced Feistel network and supports variable block and key lengths. We chose the variant with a 128-bit block and key length and 20 rounds since this version was submitted as an AES candidate and allows a fair comparison with the other ciphers. The basic operations needed to implement RC6 are bitwise logical *xor*, modular addition and subtraction, modular squaring, and data-dependent rotation. All operations are performed on 32-bit words.

Rijndael (AES)

Rijndael, introduced by Daemen and Rijmen in 1998 [5], is a substitution-permutation network with 10 rounds for the variant with the 128-bit key length that we consider. The 128 bits of a block are arranged in a 4×4 matrix of bytes where each byte is viewed as an element of the finite field GF(2^8). The round transformation can be described using arithmetic operations (addition, multiplication, inversion) in this field [5]. However, the main operations needed on our target platform are 8-bit to 8-bit table lookup, bytewise rotate, and bitwise logical *xor*.

Serpent

Serpent was proposed by Anderson et al. in 1998 [1]. It is a substitution-permutation network with 32 rounds. Serpent is designed to allow for an optimized implementation using the bitslice technique [4]. Hence, instead of table lookups, an efficient way to implement Serpent is to use operations like fixed shift/rotate and logical *and*, *or*, *xor*, and *not*.

Twofish

Twofish is a Feistel-type cipher with 16 rounds, proposed by Schneier et al. in 1998 [14]. A unique feature of Twofish are its key-dependent S-boxes based on two fixed permutations on 8-bit values, which are typically implemented via table lookup. The S-box operation is followed by a matrix multiplication over the finite field GF(2^8). Twofish also performs logical *xor*, addition mod 2^{32}, and fixed bitwise rotation.

XTEA

XTEA was introduced by Needham and Wheeler in 1997 as an improvement of the Tiny Encryption Algorithm (TEA) [10]. It operates on data blocks of 64 bits with a 128-bit key and performs 64 rounds. XTEA is characterized by a very simple round function; the only operations being used are fixed bitwise rotation, logical *xor*, and modular addition.

5. DESCRIPTION OF CODE SIZE AND MEMORY OPTIMIZATIONS

We implemented Rijndael and XTEA from scratch with minimal code size in mind. In some cases (e.g. Rijndael) it is possible to use run-time-generated lookup tables, stored in RAM, to reduce the code size. In these cases, we preferred a small memory footprint over minimized code size, as RAM is normally much more scarce in embedded systems than program ROM. The implementations of RC6, Serpent, and

Twofish are based on Brian Gladman's code [6], which are performance-optimized variants for 32-bit platforms. We took a number of measures to reduce the code size of these implementations, whereby the main point was always to replace pre-processor macros (which result in inline code) with functions calls.

Other general measure for code size reduction include the use of data types which correspond to the native word size of the target processor (32 bit) for all C variables and the consideration of the maximal size of constants or immediate values (11 bits on ARM). We used inline assembler for a few critical operations in order to take advantage of special architectural features of ARM like its built-in shift/rotate mechanism for ALU instructions. This mechanism allows to perform an optional shift of a register or immediate value "for free" as part of an ALU instruction.

RC6

Our RC6 implementation is derived from Brian Gladman's source code [6]. We replaced all macros by function calls and used rolled loops in the encrypt/decrypt functions to save code size. In order to reduce the function call overhead, we replicated the body of the loop four times, i.e. each iteration of the loop actually executes four round transformations.

Rijndael (AES)

On 32-bit platforms, Rijndael is usually implemented with round lookup (T-lookup) tables in order to maximize performance [6]. Depending on the configuration, these lookup tables have a size of between 1 kB and 8 kB for encryption and between 1 kB and 12 kB for decryption. Hence, the T-lookup tables can occupy up to 20 kB of memory in the most extreme case. However, it is possible to avoid these tables and to implement Rijndael with just two 256-byte tables for the forward S-box (encryption and key expansion) and inverse S-box (decryption), as well as 12 bytes for the round key constant table (rcon). We implemented Rijndael following this strategy and the concepts of Bertoni et al. [3] to allow for an efficient execution of the costly MixColumns and InvMixColumns transformation on 32-bit platforms.

Serpent

Again our implementation is a modification of Gladman's code [6]. This code follows the bit-slice approach proposed by the inventors of Serpent [1]. We replaced the macros by function calls, which markedly reduced the code size of the C functions implementing the S-box. The use of temporary local variables in these C functions allows for further code size reduction without notable loss of performance. We also rewrote the en/decrypt functions and brought them into a very compact form with the help of C function pointers.

Twofish

Twofish is also based on Gladman's code [6], whereby the macros are again replaced by function calls. Fixing the key size to 128 bits and implementing the en/decrypt functions with rolled loops allowed for further reduction of the code size. However, the benefit achieved by using loops is limited since the round function is executed only eight times.

XTEA

XTEA is trivial to implement. There is absolutely no room for an improvement of the original source code from [10].

5.1 Comparison of Code Size and Memory Footprint

Figure 2 summarizes the code size of the resulting block cipher implementations after we made the modifications and optimizations described before. All values are given in bytes and represent the size of the `text` segment. XTEA is the clear winner in this comparison due to its simplicity and the lack of a key expansion function. The RC6 implementation is also very small; encryption and decryption occupy less than 1 kB altogether. Twofish achieves average results in terms of code size. However, it must be considered that the Twofish code contains two 256-byte lookup tables for the 8-bit permutations q_0 and q_1 [14]. Figure 2 shows the size of Gladman's Rijndael as well as our own Rijndael implementation. Gladman's code is twice as large as ours when both encryption and decryption are needed. Serpent is also relatively large in terms of code size, primarily due to the bit-slice approach for the S-box implementation.

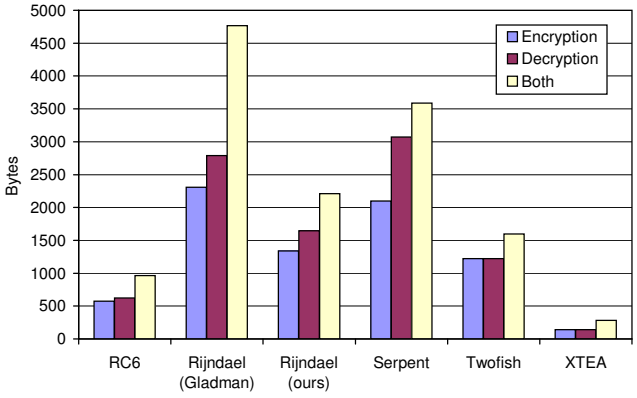

Figure 2: Comparison of code size.

All five block ciphers except Rijndael were implemented without large lookup tables in order to reduce the run-time memory footprint to a minimum. Gladman's Rijndael implementation performs the S-box operation via a 256-byte lookup table whose entries are calculated at run-time with help of the `genTabs` function [6]. Of course, two 256-byte tables are necessary if both encryption and decryption have to be performed, amounting to 512 bytes altogether. On the other hand, our Rijndael implementation uses a static lookup table for the S-box operation. In summary, Rijndael requires marginally more run-time memory than the other four ciphers. However, the overall run-time memory (RAM) utilization of each cipher is well below 1 kB, which confirms that the implementations are indeed "lightweight" and fulfill the memory constraints we imposed.

6. SIMULATION RESULTS

We simulated the five block cipher implementations with Sim-Panalyzer [18] in order to analyze and compare their execution times and energy consumption. Sim-Panalyzer provides architectural and power configuration templates for the StrongARM SA-1100 processor. All source codes were compiled, assembled, and linked with an ARM cross compiler based on the GNU GCC and the BINUTILS tool chain. We put a similar effort into optimizing each of the five block ciphers to ensure a fair comparison.

6.1 Performance

Figure 3 summarizes the running times (in clock cycles) of the key expansion and the encryption of a 128-bit data block. RC6 is by far the fastest block cipher, but suffers from a relatively costly key expansion. The two Rijndael implementations have a faster key expansion than RC6, but need more time to encrypt a 128-bit block. XTEA is roughly comparable to Rijndael and has the advantage that it does not need a key expansion. Serpent and Twofish show rather bad performance in both key expansion and encryption.

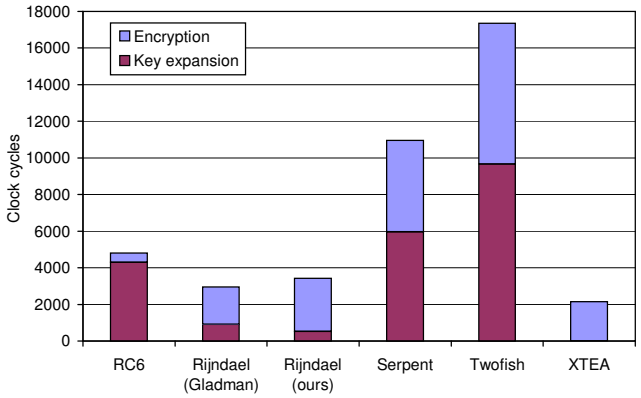

Figure 3: Performance of key expansion and encryption of a 128-bit data block.

Figure 4 illustrates the decryption performance of the five block cipher implementations. The relative performance figures are very similar to that of encryption. Again, RC6 wins big followed by XTEA and Rijndael. However, as in encryption, RC6 has the penalty of costly key expansion. A noteworthy detail is that Gladman's Rijndael code is much faster for encryption than for decryption. Our Rijndael implementation, on the other hand, has similar execution times for both encryption and decryption.

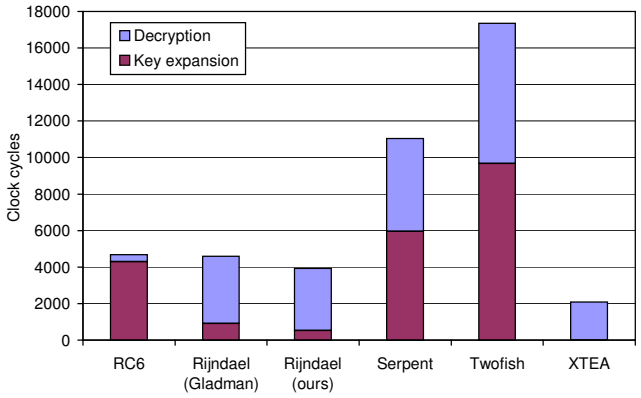

Figure 4: Performance of key expansion and decryption of a 128-bit data block.

The key expansion does not fall into account when large amounts of data are encrypted or decrypted. For "bulk encryption," a so-called *mode of operation* specifies how the individual blocks of the plaintext are encrypted and linked together. Table 2 shows the throughput of the five block ciphers for different modes of operation. We evaluated the

Cipher	ECB	CBC	CTR	CFB	OFB
RC6	5.240	4.407	4.298	4.419	4.426
Rijndael	1.436	1.365	1.354	1.366	1.367
Serpent	0.624	0.610	0.608	0.610	0.610
Twofish	0.398	0.392	0.391	0.392	0.392
XTEA	1.664	1.543	1.517	1.546	1.548

Table 2: Throughput (in MB/s) for different modes of operation at a clock frequency of 200 MHz.

throughput by encrypting a message of size 15.62 kB, which corresponds to exactly 1,000 blocks of 128 bits.

The modes of operation we considered in our study are the electronic codebook (ECB) mode, the cipher-block chaining (CBC) mode, the counter (CTR) mode, the cipher feedback (CFB) mode, and the output feedback (OFB) mode. ECB and OFB achieve the best results, but the difference to the other modes is relatively small. Regarding throughput, RC6 is again the winner followed by XTEA and Rijndael. Twofish and Serpent lag significantly behind the others.

6.2 Power/Energy Consumption

The energy consumed by a processor during the execution of a piece of software, such as a block cipher, corresponds to the product of the average power dissipation and the total running time. The former depends on a number of factors including supply voltage, clock frequency, and the average current drawn by the processor while executing individual instructions of the program code. Sinha et al. analyzed in [15] the power characteristics of the StrongARM SA-1100 and found that its overall power dissipation is dominated by a few subsystems like the cache, control logic, and clock tree. On the other hand, the functional units (e.g. ALU) on which the instructions are actually executed have almost no impact on the overall power. Therefore, the variation of the current consumption between different instructions is rather small (at most 38% of the overall average current [15]).

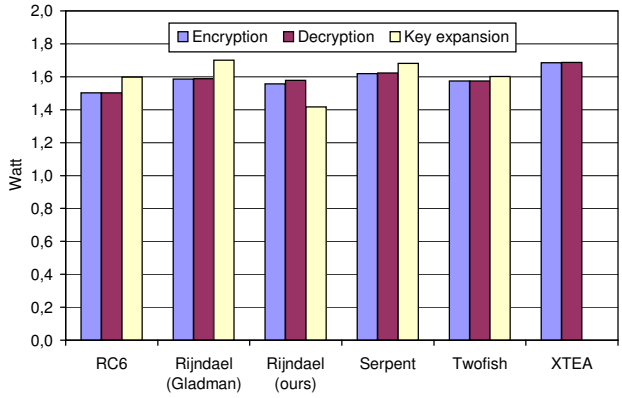

Figure 5: Average power consumption during encryption, decryption, and key expansion.

Figure 5 depicts the average power consumption of the StrongARM SA-1100 for encryption, decryption, and key expansion (at a supply voltage of 2 V and a clock frequency of 200 MHz). As expected, the average power consumption is pretty uniform amongst the different block ciphers, which confirms the observations made by Sinha et al. [15]. Due to

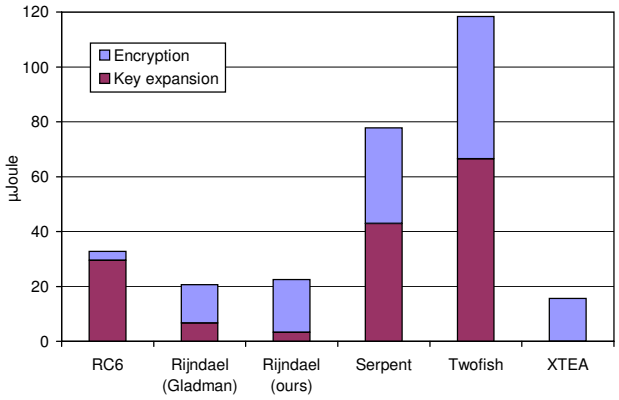

Figure 6: Processor core energy for key expansion and encryption of a 128-bit data block.

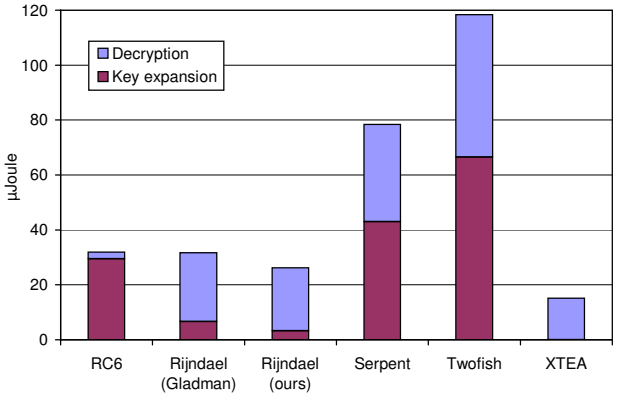

Figure 7: Processor core energy for key expansion and decryption of a 128-bit data block.

this uniform power profile, the energy consumption of the block ciphers will primarily depend on the execution time.

Figure 6 and 7 illustrate the energy consumption of the StrongARM for encryption and decryption, respectively. In addition, the energy cost of key expansion is also specified for each block cipher. As expected, the energy values are closely tied to the corresponding execution times shown in Figure 3 and 4. The reason for this close relation between energy and execution time is the uniform average power dissipation, which differs less than 20% among the different block ciphers (see Figure 5). In summary, RC6 is by far the most energy-efficient block cipher, followed by XTEA and Rijndael.

7. CONCLUSIONS

In this paper we evaluated the performance and energy-efficiency of the block ciphers RC6, Rijndael, Serpent, Twofish, and XTEA on a StrongARM processor. We conducted our evaluation with "lightweight" software implementations optimized for small code size and low memory footprint. The run-time memory utilization of each of our five block cipher implementations is less than 1 kB, leaving more resources for the main application. Our simulation results, obtained with Sim-Panalyzer using power models for the StrongARM SA-1100, show that RC6 is extremely fast and, hence, very energy-efficient for both decryption and encryption, but has the penalty of costly key expansion. Therefore, RC6 is a good candidate for the encryption of large amounts of data since in this case the costly key expansion does not fall into account. XTEA is also very fast and comes in second, after RC6, in our energy-efficiency ranking. The fact that XTEA has no key expansion makes it highly suited for application domains where short messages are encrypted, e.g. in sensor networks. Rijndael may serve as an alternative to RC6 or XTEA for medium-sized messages. Our overall conclusion is that RC6 is the fastest and most energy-efficient block cipher under memory constraints.

8. REFERENCES

[1] R. J. Anderson, E. Biham, and L. R. Knudsen. Serpent: A proposal for the Advanced Encryption Standard. Technical report, University of Cambridge, 1998.

[2] ARM Limited. ARM Architecture Reference Manual. ARM Doc No. DDI-0100, Issue H, 2003.

[3] G. Bertoni et al. Efficient software implementation of AES on 32-bit platforms. In *Cryptographic Hardware and Embedded Systems — CHES 2002*, LNCS 2523, pp. 159–171. Springer Verlag, 2002.

[4] E. Biham. A fast new DES implementation in software. In *Fast Software Encryption — FSE '97*, LNCS 1267, pp. 260–272. Springer Verlag, 1997.

[5] J. Daemen and V. Rijmen. *The Design of Rijndael: AES – The Advanced Encryption Standard*. Springer Verlag, 2002.

[6] B. R. Gladman. AES second round implementation experience. Available online at `http://fp.gladman.plus.com/cryptography_technology/aesr2/index.htm`, 2000.

[7] C. T. Hager et al. Performance and energy efficiency of block ciphers in personal digital assistants. In *Proceedings of the 3rd IEEE International Conference on Pervasive Computing and Communications (PerCom 2005)*, pp. 127–136. IEEE Computer Society Press, 2005.

[8] T. Kindberg and A. Fox. System software for ubiquitous computing. *IEEE Pervasive Computing*, 1(1):70–81, 2002.

[9] Y. W. Law, J. M. Doumen, and P. H. Hartel. Survey and benchmark of block ciphers for wireless sensor networks. *ACM Transactions on Sensor Networks*, 2(1):65–93, 2006.

[10] R. M. Needham and D. J. Wheeler. Tea extensions. Technical report, University of Cambridge, 1997.

[11] N. R. Potlapally et al. Analyzing the energy consumption of security protocols. In *Proceedings of the 8th International Symposium on Low Power Electronics and Design (ISLPED 2003)*, pp. 30–35. ACM Press, 2003.

[12] D. Remondoa and I. G. Niemegeers. Ad-hoc networking in future wireless communications. *Computer Communications*, 26(1):36–40, 2003.

[13] R. L. Rivest et al. The RC6^{TM} block cipher. Technical report, RSA Laboratories, 1998.

[14] B. Schneier et al. Twofish: A 128-bit block cipher. Technical report, Counterpane Systems, 1998.

[15] A. Sinha and A. P. Chandrakasan. JouleTrack - A web based tool for software energy profiling. In *Proceedings of the 38th Design Automation Conference (DAC 2001)*, pp. 220–225. ACM Press, 2001.

[16] F. Stajano. *Security for Ubiquitous Computing*. John Wiley and Sons Ltd, 2002.

[17] S. Tillich, J. Großschädl, and A. Szekely. An instruction set extension for fast and memory-efficient AES implementation. In *Communications and Multimedia Security — CMS 2005*, LNCS 3677 pp. 11–21. Springer Verlag, 2005.

[18] University of Michigan. *Sim-Panalyzer 2.0*. Available for download at `http://www.eecs.umich.edu/~panalyzer`.

[19] M. Weiser. The computer for the 21st century. *Scientific American*, 265(3):94–104, 1991.