Benchmarking Block Ciphers for Wireless Sensor Networks (Extended Abstract)

Yee Wei Law Jeroen Doumen Pieter Hartel
Faculty of Electrical Engineering, Mathematics and Computer Science
University of Twente, The Netherlands
Email: {ywlaw, doumen, pieter}@cs.utwente.nl

Abstract

The energy efficiency requirement of wireless sensor networks (WSNs) is especially high because the sensor nodes are meant to operate without human intervention for a long period of time with little energy supply. Besides, available storage is scarce due to their small physical size. Therefore choosing the most storage- and energy-efficient block cipher for WSNs is important. However to our knowledge so far, no systematic work has been conducted in this area. In this paper, we have identified the candidates of block ciphers suitable for WSNs based on existing literature and authoritative recommendations. We have benchmarked these candidates and based on this benchmark, we have selected the suitable ciphers for WSNs, namely Rijndael for high security and energy efficiency requirements; but MISTY1 for good storage and energy efficiency. In terms of operation mode, we recommend Output Feedback Mode for static networks, but Counter Mode for dynamic networks.

1. Introduction

A wireless sensor network (WSN) is a network composed of a large number of sensors that (1) are physically small, (2) communicate wirelessly among each other, and (3) are deployed without prior knowledge of the network topology. Due to the limitation of their physical size, the sensors tend to have storage space, energy supply and communication bandwidth so limited that every possible means of reducing the usage of these resources is aggressively sought. For example, a sensor typically has 8∼120KB of code memory and 512∼4096 bytes of data memory. The energy supply of a sensor is such that it will be depleted in less than 3 days if operated constantly in active mode [18]. The transmission bandwidth ranges from 10kbps to 115kbps. Table 1 compares the sensor node used in the EYES project (eyes.eu.org) with Smart Dust [12] and the Intel Research mote [15].

Karlof and Wagner made an interesting observation that WSNs will more likely ride Moore's Law *downward* [14], that is, instead of relying on the computing power to double every 18 months, we are bound to seek ever cheaper solutions. However looking at the current development of WSNs (Table 1), computing power is indeed increasing, though not necessarily at the rate predicted by Moore's Law. Either way, we are conservative and assume that the hardware constraints of WSNs will remain rather constant for some time to come.

Table 1. Comparison of the EYES node with Smart Dust and the Intel Research mote.

	Smart Dust	EYES node	Intel mote
CPU	8-bit, 4 MHz	16-bit, 8 MHz	16-bit, 12 MHz
Flash mem.	8 KB	60 KB	512 KB
RAM	512 B	2 KB	64 KB
Frequency	916 MHz	868.35 MHz	900 MHz
Bandwidth	10 kbps	115.2 kbps	100 kbps

Cryptographic algorithms are an essential part of the security architecture of WSNs, using the most efficient and sufficiently secure algorithm is thus an effective means of conserving resources. By 'efficient' in this paper we mean requiring little storage and consuming little energy. Although transmission consumes more energy than computation, our focus in this paper is on computation and we can only take transmission energy into account when considering the security scheme as a whole. The essential cryptographic primitives for WSNs are block ciphers, message authentication codes (MACs) and hash functions. Among these primitives, we are only concerned with block ciphers in this paper, because MACs can be constructed from block ciphers [25], and hash functions are relatively cheap [6]. Meanwhile, public-key algorithms are well-known to be prohibitively expensive [4].

Our selection of block ciphers is RC5 [26], RC6 [27], Rijndael [8], MISTY1 [19], KASUMI [1] and Camellia [3]. Although Rijndael has been selected by the American National Institute of Standards and Technology (NIST) as the Advanced Encryption Standard (AES), after a five-

year long standardisation process that included extensive benchmarking on a variety of platforms ranging from smart cards [11] to high end parallel machines [32], the selection of Rijndael for our platform is *not* obvious. This is because the fact that Rijndael is *on average* the best performer on a range of standard platforms, does not mean that it also performs best on our platform, which is Texas Instruments' 16-bit RISC-based MSP430F149 [30], chosen for its ultra-low power consumption (`www.ti.com`). During the selection process of the AES, the focus has been on 8-bit smart cards, 32-bit mainstream architectures and 64-bit high-end platforms, but not 16-bit architectures, further highlighting the importance of our study.

The contribution of this paper is two-fold: (1) to measure the candidates of block ciphers suitable for WSNs, and (2) to select the suitable ciphers based on the benchmark.

The paper is organised as follows. Section 2 explains how we have arrived at our selection of block ciphers. Section 3 details aspects of our benchmarking. Section 4 provides our observation and evaluation results. Section 5 concludes.

2. Selection Rationales

We have chosen our benchmark candidates based on the following rationales:

RC5 is a well-known algorithm that has been around since 1995 without crippling weaknesses. Although `distributed.net` has managed to crack a 64-bit RC5 key in RSA Laboratories' Secret-Key Challenge after 1757 days of computing involving 58,747,597,657 distributed work units, the standard key length of RC5 is 128 bits and RC5 has managed to withstand years of cryptanalysis.

RC6, like RC5, is parameterised and has a small code size. RC6 is one of the five finalists that competed in the AES challenge and according to some AES evaluation reports [22][32], it has reasonable performance. Lastly, RC6 has been chosen as the algorithm of choice by Slijepcevic et al. [29] for WSNs. We are interested in seeing if their choice is justified.

Rijndael is the *de facto* Advanced Encryption Standard, mandated by the NIST of the United States, chosen after extensive scrutiny and performance evaluation (`csrc.nist.gov/encryption/aes`). It is also one of the ciphers recommended by the New European Schemes for Signature, Integrity and Encryption (NESSIE) Consortium (`www.cryptonessie.org`), and Japan's CRYPTREC [7]. Rijndael is well studied and there are efficient implementations on a wide range of platforms (`www.rijndael.com`). We would however like to obtain first-hand experience of evaluating Rijndael on our particular platform, which has never been studied before during the selection process of the AES.

MISTY1 is one of the CRYPTREC-recommended 64-bit ciphers [7] and is the predecessor of KASUMI, the 3GPP-endorsed encryption algorithm [1]. MISTY1 is also a royalty-free open standard documented in RFC2994 [23]. We found MISTY1 to be particularly suitable for 16-bit platforms.

KASUMI, as the 3GPP-endorsed encryption algorithm [1], is presumably well-suited for embedded applications, and has gone through considerable expert scrutiny.

Camellia is one of the NESSIE- and CRYPTREC-recommended 128-bit ciphers [7]. Like MISTY1, Camellia is also royalty-free. Security-wise, Camellia has been designed with state-of-the-art modern techniques with a large safety margin in view of anticipated progress in cryptanalysis techniques [2][20]. We are interested in seeing how it performs on our platform.

3. Methodology and Consideration

For benchmarking, we consider: (1) the cipher parameters, (2) the cipher operation modes, (3) the implementation sources, and (4) the compiler toolchain.

3.1. Cipher Parameters

Table 2 lists the parameters we have adopted for each cipher (some of them actually have fixed, unadjustable parameters but we list them anyway for clarity's sake). The number of rounds for each cipher is nominal except for RC5, where 18 is used instead of the nominal 16, for security reasons described in our technical report [17]. Although RC5 and RC6 allow variable word size, without the backing of relevant cryptanalytic research, we are not sure how many rounds to use if we pick a non-standard word size of 16 bits, which is the word size of our platform. Therefore, for RC5 and RC6, we are using the standard word size of 32 bits.

Table 2. Cipher parameters (lengths in bytes).

Cipher	Key length	Rounds	Block length	Expanded key length
RC5	16	18	8	152
RC6	16	20	16	176
Rijndael	16	10	16	240
MISTY1	16	8	8	64
KASUMI	16	8	8	128
Camellia	16	18	16	272

3.2. Operation Modes

The naïve approach of encrypting a message longer than one block, by dividing the message into multiple blocks

and encrypting the blocks separately, is called the *electronic codebook mode* (ECB) mode. ECB is insecure since an adversary can construct valid ciphertexts from the original ciphertext by arbitrarily rearranging, repeating and/or omitting blocks from the original ciphertext. More secure operation modes in Table 3 are used in practice. These operation modes do not only affect the security, but also the energy-efficiency of the encryption scheme. Their fault tolerance against ciphertext error (where ciphertext bits are changed during transmission), and synchronisation error (where whole ciphertext blocks are lost) must also be taken into account. In our implementation of CBC, ciphertext stealing [28] is supported, so padding is not required. Meanwhile, some researchers [16] claim that CBC has an information leakage vulnerability due to the birthday paradox. For CFB and OFB, we are using a feedback size that is equal to the block size.

Table 3. Comparison of operation modes.

Operation mode	On ciphertext error...	On synchronisation error...
Cipher-Block Chaining (CBC)	An erroneous bit affects the entire current block and the corresponding bit of the next block.	Affected block needs to be re-transmitted to decrypt the next block.
Cipher Feedback Mode (CFB)	An erroneous bit affects the corresponding bit of the current block and the entire next block.	Affected block needs to be re-transmitted to decrypt the next block.
Output Feedback Mode (OFB)	An erroneous bit affects the corresponding bit of the current block.	Affected block does not need to be re-transmitted.
Counter (CTR)	Similar to OFB.	Similar to OFB.

3.3. Implementation Sources

To avoid reinventing the wheel, we try to use and improve as much existing source code as possible. We briefly compare a few source code libraries that we are aware of in Table 4. We have adapted most of our code from OpenSSL, and for the ciphers that do not have public implementation, we have adapted the reference source code from the original papers the ciphers are proposed in. Our source code and benchmark results can be found at `http://wwwes.cs.utwente.nl/eyes/crypto_test.zip`.

3.4. Compilers

For compilation, we are currently only using IAR Systems' MSP430 C Compiler V2.20A/W32 (`www.iar.com`). For debugging and profiling, we use IAR Systems' Embedded Workbench 3.2 with the integrated C-Spy Debugger and profiler plug-in. Another viable compiler is the GNU C compiler in the MSPGCC toolchain (`mspgcc.sf.net`), however we are not using it due to the lack of profiling support by the toolchain. That said, we do not rule out the possibility of performing our benchmarks using the toolchain as it continues to mature in the future. Note that the chip supplier itself Texas Instruments offers only the Kickstart version of the toolchain we are using.

Table 4. Comparison of several cryptographic libraries.

Sources	Advantages	Disadvantages
OpenSSL (`www.openssl.org`)	(1) Its cipher implementations are widely believed to be highly optimised with the help of the cipher designers themselves and after years of tweaking. (2) It uses low-level C, avoiding the overhead that would have been induced by higher-level languages like C++. (3) It supports most if not all cipher operation modes.	(1) It only implements standard ciphers such as the DES, RC4, RC5, Rijndael and no experimental modern ciphers (e.g. RC6, MISTY1, KASUMI, Camellia). (2) The cipher implementations do not present a uniform interface, like other C++ libraries do.
Crypto++ (`www.eskimo.com/~weidai/cryptlib.html`)	(1) This free C++ class library of cryptographic schemes by Wei Dai, supports most if not all cipher operation modes. (2) The binaries of Crypto++ version 5.0.4 have received FIPS 140-2 level 1 validation. (3) It is well-suited for benchmarking.	(1) By using C++, it incurs performance overhead and porting issues for embedded platforms. (2) Although it does implement some non-standard modern ciphers (e.g. RC6), it does not implement many others (e.g. MISTY1, KASUMI and Camellia).
Botan (`botan.randombit.net`)	(1) Supports most if not all cipher operation modes. (2) It is well-suited for benchmarking.	Similar to Crypto++.
Catacomb (`www.excessus.demon.co.uk/misc-hacks`)	(1) Supports most if not all cipher operation modes. (2) It is well-suited for benchmarking. (3) Compared with Crypto++ and Botan, it uses faster and more portable C.	Although it does implement some non-standard modern ciphers (e.g. RC6), it does not implement many others (e.g. MISTY1, KASUMI and Camellia).

In our benchmarks, we compare maximum size optimisation with maximum speed optimisation. The IAR compiler supports 3 levels of optimisation in terms of size or speed: High, Medium and Low, as well as 4 kinds of transformations: common subexpression elimination (`ELIM`), loop unrolling (`UNROLL`), function inlining (`INLINE`) and code motion (`MOTION`). Table 5 lists the levels of optimisation and kinds of transformations we use for each cipher. Furthermore, all code is compiled to use the hardware multiplier. In Table 5 and other tables henceforth, key setup modules, encryption modules, decryption modules and lookup tables are labelled as `skey`, `enc`, `dec`

and `tab` respectively. For MISTY1 and KASUMI, the IAR compiler has trouble applying Medium/High size/speed optimisation as well as code transformations on `enc` and `dec`, hence only Low optimisation is used. For all key setup algorithms, `ELIM` and `MOTION` help reduce the code size further than using size optimisation alone. `ELIM` but not `MOTION` has the same effect on CBC, but not on CFB, OFB and CTR.

Table 5. Optimisations and transformations.

Module	For size	For speed
skey, enc*, dec*	High, ELIM, MOTION	High, ELIM, UNROLL, INLINE, MOTION
CBC	High, ELIM	High, ELIM, UNROLL, INLINE, MOTION
tab, CFB, OFB, CTR	High	High, ELIM, UNROLL, INLINE, MOTION

* Except for MISTY1 and KASUMI, which use only Low optimisation and no transformation for both size and speed.

4. Results

We have performed our measurements in standalone mode, i.e. without interaction with an OS. We have taken care in making the interface of the cipher implementations as uniform as possible, so that no difference in performance is a result of the difference in the interfaces. Our benchmark parameters are memory and CPU cycles. Given in Section 4.1 and Section 4.2 are the results in terms of these two parameters, followed immediately by our observation and analysis in Section 4.3.

4.1. Memory

We refer to two types of memory: (1) code memory, in the form of Flash memory and (2) data memory, in the form of RAM. The memory organisation of MSP430F149 is such that data memory ranges from address 0200h to 09FFh, whereas code memory ranges from 01100h to 0FFFFh. The IAR compiler generates 3 types of segments for MSP430: CODE, CONST and DATA. A typical policy is to put CODE and CONST segments in the code memory, while DATA segments in the data memory. Each segment type is further divided into the following sub-types:

1. CODE: CODE (program code), CSTART and INTVEC;
2. CONST: DATA16_AC, DATA16_C (constants including string literals), DATA16_ID (initial values of static and global variables) and DIFUNCT;
3. DATA: CSTACK (the C runtime stack), DATA16_AN, DATA16_I (initialised variables), DATA16_N, DATA16_Z and HEAP.

Of all the above segment sub-types, we use only the underlined ones. The memory usage of these segments can be read off the list files generated by the compiler, and the results are shown in Table 7 and 6.

In Table 6, notice that Rijndael has two figures for the data memory of `skey`: 16 is for encryption key setup, whereas 32 is for decryption key setup. Other ciphers use only one key setup algorithm for both encryption and decryption.

In Table 7, the code memory for each CBC module takes into account (1) the code memory for key setup, (2) the code memory for barebone encryption and decryption, (3) the code memory for lookup tables, as well as (4) the code memory for CBC-specific parts. CFB, OFB and CTR do not use the decryption function [28], hence the code memory for each CFB/OFB/CTR module is similarly calculated except that the code memory for decryption, and decryption key setup in Rijndael's case, are not included. Note that the storage for plaintext, ciphertext, cipher key as well as expanded key is not included in Table 6 nor Table 7.

Table 6. Data memory requirements.

Size optimised:

	RC5	RC6	Rijndael	MISTY1	KASUMI	Camellia
skey	92	62	16,32	4	58	170
CBC	64	100	92	62	62	148
CFB	42	62	54	40	40	110
OFB	42	62	54	40	40	110
CTR	44	64	56	42	42	112

Speed optimised:

	RC5	RC6	Rijndael	MISTY1	KASUMI	Camellia
skey	288	62	16,32	4	36	184
CBC	64	102	94	64	66	150
CFB	42	62	54	40	42	110
OFB	42	62	54	40	42	110
CTR	44	64	56	42	44	112

Table 7. Code memory requirements.

Size optimised:

Mode	RC5	RC6	Rijndael	MISTY1	KASUMI	Camellia
CBC	1746	2576	14716	7132	9702	19708
CFB	908	1324	12688	4222	5446	12382
OFB	836	1252	12616	4150	5374	12310
CTR	986	1402	12766	4300	5524	12460

Speed optimised:

Mode	RC5	RC6	Rijndael	MISTY1	KASUMI	Camellia
CBC	6312	2878	15842	8492	10348	29206
CFB	3416	1354	12940	5052	5756	17148
OFB	3336	1274	12860	4972	5676	17068
CTR	3414	1352	12938	5050	5754	17146

4.2. CPU Cycles

The computational complexity of an algorithm translates directly to its energy consumption. Assuming the energy per CPU cycle is fixed (which is justified in the Appendix), then by measuring the number of CPU cycles executed per byte of plaintext processed, we get the amount of energy consumed per byte. For example MSP430F149 draws a nominal current of 420μA at 3V and at a clock frequency of 1MHz in active mode [30] – this means that the energy per instruction cycle (*for the processor alone*) is theoretically 1.26 nJ.

To evaluate the ciphers, we must determine the range of plaintext lengths that is of greatest interest to sensor networks. Here is how we arrive at the choice of 4 to 64 bytes: first we would like to see the effect of half-blocks on the ciphers' performance since we expect the ciphers to be less efficient with partial blocks, so we start with the half-block length of the 64-bit ciphers (RC5, MISTY1, KASUMI), that is 4 bytes. We then look at some MAC protocols like Sensor-MAC or S-MAC [33]. In S-MAC, the control packets are around 10 bytes long, and the data packet header is 13 bytes. Meanwhile Perrig et al. [24] suggest that data packets of 30 bytes long are realistic. Doubling this length (30 bytes) for the worst case scenario, we get 60 bytes, and rounding it off to a whole-block length, we get 64 bytes. Hence we think 4 to 64 bytes with an interval of 4 bytes is a reasonable range for our benchmarks.

Figure 1a shows our measurements for CBC encryption. CBC decryption consumes a slightly different number of CPU cycles, but the relative ordering between the various ciphers remains the same. Of particular interest is that when size-optimised, MISTY1 is more efficient than when it is speed-optimised, and it actually outperforms size-optimised Rijndael when the plaintext is shorter than 8 bytes. At more than 8 bytes, Rijndael becomes more efficient, partly because in that case it takes fewer invocations of the Rijndael encryption function than it takes the MISTY1 encryption function. Figure 1b shows that speed-optimised MISTY1 is more efficient than speed-optimised Camellia, for plaintext lengths of below 8 bytes, and between 20 and 24 bytes.

For CFB, OFB and CTR with the exception of RC5 and RC6, we found that the number of CPU cycles consumed per byte, y, can be approximated by

$$y \approx \frac{C_f + xC_b + \lceil \frac{x}{B} \rceil C_B}{x} \quad (1)$$

where x is the plaintext length, B is the cipher's block length (both in bytes); C_f, C_b and C_B are constants. In Equation 1, the first term C_f accounts for the function call overhead, the second term xC_b accounts for the overhead of organising x bytes into B-byte blocks, and the last term accounts for the actual en/decryption process and in CTR's

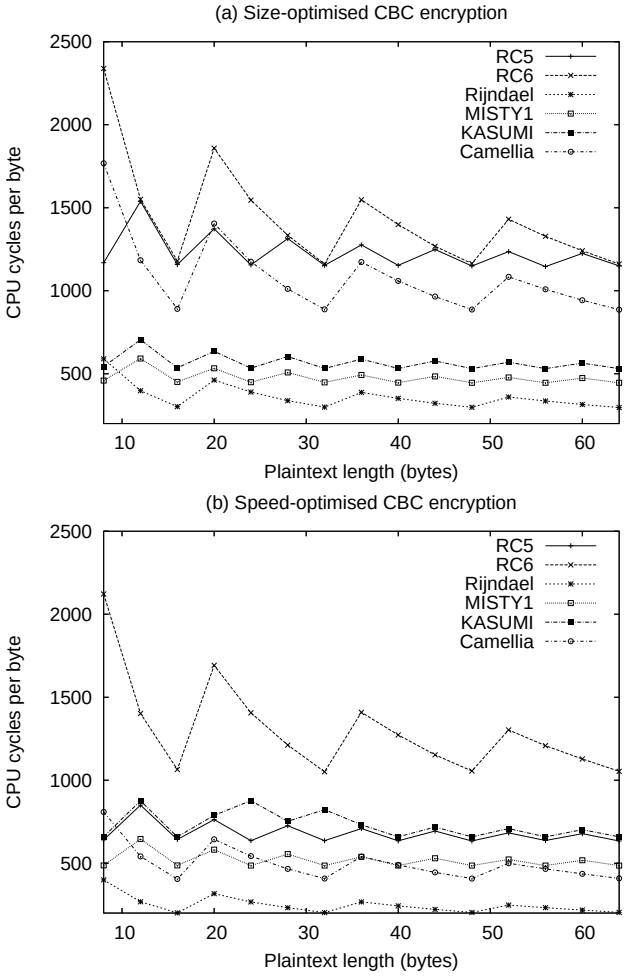

Figure 1. CPU usage of CBC encryption when (a) size-optimised, and (b) speed-optimised.

case the increment of a counter. This approximation does not apply to CBC because with ciphertext stealing CBC involves more complicated computation. This approximation does not apply to RC5 and RC6 either because an RC5/RC6 en/decryption executes a different number of rotations depending on the data and the key, resulting in a non-constant value for C_B. Thus for RC5 and RC6, the following equation is more appropriate:

$$y \approx \frac{C_f + xC_b + \lceil \frac{x}{B} \rceil V_B}{x} \quad (2)$$

where V_B is a variable. Table 8 lists the values of C_f, C_b, C_B (for Rijndael, MISTY1, KASUMI and Camellia) and V_B (for RC5 and RC6) obtained through least squares fitting. The estimated values of V_B are given in ranges. The standard error of each of the constants is less than 10^{-12}. Note that CFB, OFB and CTR consume exactly the same

number of CPU cycles for both encryption and decryption.

Table 8. Values of C_f, C_b, C_B (or V_B) for CFB, OFB and CTR.

Mode	Cipher	Size optimised			Speed optimised		
		C_f	C_b	C_B or V_B	C_f	C_b	C_B or V_B
CFB	Rijndael	94	51	3717	86	35	2613
	Camellia	94	51	11949	86	35	5885
	RC6	94	51	17561-17671	86	35	16201-16361
	MISTY1	94	51	3171	86	35	3531
	KASUMI	94	51	3849	86	35	4912
	RC5	94	51	8941-8991	86	35	4751-4861
OFB	Rijndael	82	27	3717	82	27	2613
	Camellia	82	27	11949	82	27	5885
	RC6	82	27	17561-17671	82	27	16201-16361
	MISTY1	82	27	3171	82	27	3531
	KASUMI	82	27	3849	82	27	4912
	RC5	82	27	8941-8991	82	27	4751-4861
CTR	Rijndael	90	33	3774	90	33	2662
	Camellia	90	33	12006	90	33	5934
	RC6	90	33	17618-17728	90	33	16250-16410
	MISTY1	90	33	3214	90	33	3574
	KASUMI	90	33	3892	90	33	4955
	RC5	90	33	8984-9034	90	33	4794-4904

In Table 8, it can be easily seen that OFB is the fastest mode both when size- and speed- optimised. Although not apparent in Figure 1 and Table 8, CTR is more efficient than CBC as shown in Figure 2 using Rijndael as an example. All in all, the speed comparison is OFB $>$ CTR $>$ CFB $>$ CBC when size-optimised, but OFB $>$ CFB $>$ CTR $>$ CBC when speed-optimised. Observe also that size optimisation only speeds up CBC and CFB, but not OFB and CTR.

Apart from en/decryption, we are also interested in the efficiency of the key setup algorithms. Table 9 has the results. Note again that encryption key setup and decryption key setup consume different numbers of CPU cycles in Rijndael's case.

Table 9. CPU cycles for key setup (per key).

Opt.	RC5	RC6	Rijndael	MISTY1	KASUMI	Camellia
Size	81804	96650	1745, 6769	1057	2564	23211
Speed	40556	93811	1313, 5034	584	1296	15335

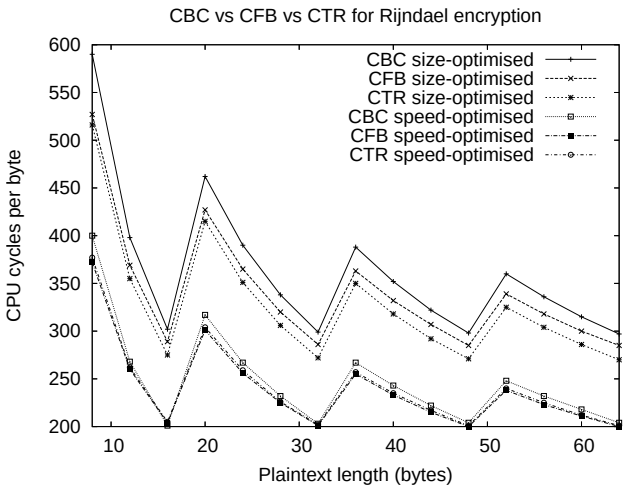

Figure 2.

4.3. Observation and Analysis

First about the operation mode. Apart from having the highest memory and energy efficiency, OFB has desirable fault-tolerance characteristics especially in an error-prone environment such as the wireless network, that is (1) an erroneous ciphertext bit affects only the corresponding bit of the decrypted plaintext block, (2) the decryption of a ciphertext block is independent of the previous block (cf Table 3). However in the event that multiple blocks are lost, for example if i blocks c_1, ..., c_i are lost, the computing cost of the $(i+1)$-th plaintext block using OFB is $i+1$ encryptions of the initialisation vector, which is higher than the corresponding cost using CTR, involving just 1 increment and 1 encryption of the counter.

Next we analyse Figure 1, Table 7, 6 and 8 cipher by cipher:

RC5: The speed-optimised version is 2 times as fast as the size-optimised version, at a price of 50% more code size and 3 times as much data size (which is 14% of the available RAM in an EYES node).

RC6: Nechvatal et al. [22] note that on the Z80 processor, the key setup of RC6 is very time-consuming, and it takes about 4 times as many cycles as encryption (of one block) does. Our measurements reveal that key setup takes at least 5 times as many cycles as CBC encryption (the least efficient mode) does. For en/decryption, there is no significant difference between the size-optimised and the speed-optimised version in terms of memory and CPU cycles.

Rijndael: Rijndael has a large S-box, eating up over 10KB of code memory, which is about the current size of the EYES operating system [21]. Among our selection, Rijndael is the only cipher that has a separate key setup for decryption, and we note that this key setup is about 4 times

as slow as that for encryption.

MISTY1: As mentioned, except for key setup, MISTY1 is actually more efficient when size-optimised than when speed-optimised. The size-optimised version also provides 17-19% savings in storage, with largely the same data memory requirement. MISTY1 requires only 4 bytes of data memory for key setup.

KASUMI: Like MISTY1, the size-optimised version of KASUMI is more efficient than the speed-optimised version. Although the key setup of KASUMI is linear [9], it takes more than 2 times as long as that of MISTY1 which is non-linear. Compared with MISTY1, KASUMI uses more memory and is less efficient.

Camellia: Camellia has the largest code and data memory requirement, but with speed optimisation, Camellia is only worse than Rijndael in efficiency. This observation coincides with Granelli et al.'s conclusion that "Camellia is close to Rijndael" in performance [10].

To conclude our observations, we now discuss the rankings of the ciphers in Table 10.

Table 10. Ranking of ciphers*.

En/Decryption			Key setup		
Code memory	Data memory	Speed	Data memory	Speed	Expanded key storage
$RC5_z$	MISTY1, $KASUMI_z$	$Rijndael_s$	MISTY1	$MISTY1_s$	MISTY1
$RC6_z$	$KASUMI_s$, RC5	$Rijndael_z$	Rijndael	$MISTY1_z$	KASUMI
$RC6_s$	Rijndael	$Camellia_s$	KASUMI	$KASUMI_s$	RC5
$RC5_s$	RC6	$MISTY1_z$	RC6	$KASUMI_z$	RC6
$MISTY1_z$	Camellia	$MISTY1_s$	$RC5_z$	$Rijndael_s$	Rijndael
$MISTY1_s$		$KASUMI_z$	$Camellia_z$	$Rijndael_z$	Camellia
$KASUMI_z$		$RC5_s$	$Camellia_s$	$Camellia_s$	
$KASUMI_s$		$KASUMI_s$	$RC5_s$	$Camellia_z$	
$Camellia_z$		$Camelia_z$		$RC5_s$	
$Rijndael_z$		$RC6_s$		$RC5_z$	
$Rijndael_s$		$RC6_z$		$RC6_s$	
$Camellia_s$		$RC5_z$		$RC6_z$	

* Subscript z means size-optimised, s means speed-optimised.

In terms of en/decryption, RC5 scores high on (meaning 'requires little') code memory and data memory while low on speed. While RC6 excels in small code size, it is the slowest even after speed optimisation. Speed-optimised Rijndael offers the highest speed but has a large code size. Size-optimised MISTY1 is a solid performer in all categories, only worse than RC5, RC6 in code size, and Rijndael, Camellia in speed. KASUMI is always slightly behind MISTY1 in all categories. Speed-optimised Camellia has decent performance but the largest code size and RAM footprint.

In terms of key setup, RC5 tends to rank near the bottom except in the storage requirement of expanded keys. RC6 is only better than RC5 in terms of data memory requirement. Rijndael requires little data memory but is slower than MISTY1 and KASUMI. Rijndael also requires a lot of storage for expanded keys, but in a degree not worse than Camellia. MISTY1 is the winner, with KASUMI lagging slightly behind in all categories. Camellia requires a lot of data memory and storage for expanded keys, although it is better than RC5 and RC6 in energy efficiency. Note that expanded keys can be stored in the code or data memory.

5. Conclusion

First on the suitable operation mode to use. Although OFB is the most storage- and energy-efficient mode, in cases where multiple blocks are lost, it is less efficient than CTR for regaining synchronisation. Although speed-optimised CFB is more efficient than CTR, it requires lost blocks to be re-transmitted. CBC not only requires lost blocks to be re-transmitted, but also requires a lot of code, data memory and CPU cycles. CBC is popular in the wired world because unlike other modes, the initialisation vector can be re-used. Our conclusion is that OFB should be used in static networks while CTR in dynamic networks, assuming static networks are less prone to packet loss. Parallel to our conclusion, Perrig et al. have used the CTR mode in their SPINS testbed [24].

On the suitable cipher to use, we will reach our verdict by first ruling out the unlikely candidates. First we would like to emphasise that MSP430F149 is one of the most high-end in the Texas Instrument's MSP430 series. In other words, a total code memory of 59.7KB and data memory of 2KB is a hard limit in the MSP430 family of processors. For this reason, we first rule out Camellia which occupies 1/5 of the total code memory even after size optimisation, even though it has good energy-efficiency for en/decryption when speed-optimised. The next to be ruled out are RC5 and RC6, which have poor energy efficiency and key agility (the ability to change keys quickly and with a minimum amount of resources). KASUMI lags behind MISTY1 in all categories, so we should consider MISTY1 instead of KASUMI.

Finally the verdict: for maximum energy-efficiency, we recommend speed-optimised Rijndael; whereas for truly constrained environments, we recommend size-optimised MISTY1. Although MISTY1 is slower than Rijndael in en/decryption, MISTY1 has higher key agility and requires less memory (e.g. MISTY1's code size is 1/3 of Rijndael's). One other drawback apart from en/decryption speed is that MISTY1 is believed to be less secure than Rijndael and it only caters for one key length, but we foresee that there are applications that do not require a security level higher than MISTY1 can provide, such as environmental sensing.

Reviewing some of the proposals in the literature so far, we see that we are disagreeing with Perrig et al. [24] about

using RC5, and with Slijepcevic et al. [29] about using RC6. Another aspect to consider is that most embedded processors do not support the variable-bit rotation instruction like ROL of the Intel architecture [13] which greatly improves the performance of RC5. Slijepcevic et al. suggest varying the number of rounds to achieve different levels of security with RC6, but increasing the number of rounds beyond the nominal value to increase the level of security is a speculative approach compared with increasing the key length. That is, using longer keys for higher security is a more appropriate approach. One non-technical reason against the use of RC6 is that the cipher is patented and not royalty-free.

In conclusion, we have presented a detailed benchmark for one of the most important cryptographic primitives for WSNs, i.e. block ciphers. Taking into account the security properties, storage- and energy-efficiency of a set of candidates, we have arrived at a systematically justifiable selection of block ciphers and operation modes.

References

[1] 3GPP. Specification of the 3GPP Confidentiality and Integrity Algorithms Document 2: KASUMI Specification. ETSI/SAGE Specification Version: 1.0, Dec 1999.

[2] K. Aoki, T. Ichikawa, M. Kanda, M. Matsui, S. Moriai, J. Nakajima, and T. Tokita. Camellia: A 128-Bit Block Cipher Suitable for Multiple Platforms. In D. Stinson and S. Tavares, editors, *Proc. Selected Areas in Cryptography (SAC'00)*, number 2012 in LNCS, pages 39–56. Springer-Verlag, 2001.

[3] K. Aoki, T. Ichikawa, M. Kanda, M. Matsui, S. Moriai, J. Nakajima, and T. Tokita. Specification of Camellia – A 128-Bit Block Cipher. Specification Version 2.0, Nippon Telegraph and Telephone Corporation and Mitsubishi Electric Corporation, 2001.

[4] D. Carman, P. Kruus, and B. Matt. Constraints and approaches for distributed sensor network security. Technical Report #00-010, NAI Labs, 2000.

[5] P. Chien and V. Wen. CS199 – StrongARM Energy Measurement Report. Online slides: `http://www.cs.berkeley.edu/~vwen/strongarm/slides/cs199.ppt`, 1998.

[6] S. Crosby and D. Wallach. Denial of service via algorithmic complexity attacks. In *12th USENIX Security Symposium*, pages 29–44. USENIX Association, 2003.

[7] CRYPTREC. 電子政府推奨暗語の仕様書(trans.: Specification of e-government-recommended ciphers). Web page, Dec. 2003.

[8] J. Daemen and V. Rijmen. AES Proposal: Rijndael, 1999.

[9] O. Dunkelman. Comparing MISTY1 and KASUMI. NESSIE Public Report NES/DOC/TEC/WP5/029/a, Computer Science Department, Technion, Dec. 2002.

[10] F. Granelli and G. Boato. A novel methodology for analysis of the computational complexity of block ciphers: Rijndael, Camellia and SHACAL-2 compared. Technical Report DIT-04-004, University of Trento, 2004.

[11] G. Hachez, F. Koeune, and J.-J. Quisquater. cAESar results: Implementation of Four AES Candidates on Two Smart Cards. In *2nd AES Candidate Conference (AES2)*, Oct. 2000.

[12] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. *SIGOPS Oper. Syst. Rev.*, 34(5):93–104, 2000.

[13] Intel Corporation. *Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference*, 1997.

[14] C. Karlof and D. Wagner. Secure routing in wireless sensor networks: Attacks and countermeasures. *Elsevier's Ad Hoc Networks Journal, Special Issue on Sensor Network Applications and Protocols*, 1(2–3):293–315, 2003.

[15] R. Kling. Intel mote: An Enhanced Sensor Network Node. In *International Workshop on Advanced Sensors, Structural Health Monitoring and Smart Structures*, 2003.

[16] LASEC. Information leakage in electronic transactions. Web page, 2004.

[17] Y. Law, J. Doumen, and P. Hartel. Survey and benchmark of block ciphers for wireless sensor networks. Technical Report TR-CTIT-04-07, Centre for Telematics and Information Technology, University of Twente, The Netherlands, Jan. 2004.

[18] Y. Law, S. Dulman, S. Etalle, and P. Havinga. Assessing security-critical energy-efficient sensor networks. Technical Report TR-CTIT-02-18, Centre for Telematics and Information Technology, University of Twente, The Netherlands, July 2002.

[19] M. Matsui. New Block Encryption Algorithm MISTY. In E. Biham, editor, *Fast Software Encryption, 4th International Workshop, FSE '97*, volume 1267 of *LNCS*, pages 54–68. Springer-Verlag, 1997.

[20] M. Matsui and T. Tokita. MISTY, KASUMI and Camellia Cipher Algorithm. *Mitsubishi Electric ADVANCE (Cryptography Edition)*, 100:2–8, Dec 2000.

[21] J. Mulder. PEEROS: PreEmptive Eyes Real-Time Operating System. Master's thesis, University of Twente, Apr. 2003. `http://wwwhome.cs.utwente.nl/~hoesel/EyesOS/PeerOS\%20Report-Job\%20Mul%der.pdf`.

[22] J. Nechvatal, E. Barker, L. Bassham, W. Burr, M. Dworkin, J. Foti, and E. Roback. Report on the Development of the Advanced Encryption Standard (AES). Technical report, NIST, 2000.

[23] H. Ohta and M. Matsui. A Description of the MISTY1 Encryption Algorithm. RFC 2994, Network Working Group, IETF, Nov. 2000.

[24] A. Perrig, R. Szewczyk, V. Wen, D. Culler, and J. Tygar. SPINS: Security Protocols for Sensor Networks. In *Proceedings of the 7th Ann. Int. Conf. on Mobile Computing and Networking*, pages 189–199. ACM Press, 2001.

[25] B. Preneel. Cryptographic primitives for information authentication - state of the art. In B. Preneel and V. Rijmen, editors, *State of the Art in Appplied Cryptography*, volume 1528 of *LNCS*, pages 50–105. Springer-Verlag, 1998.

[26] R. Rivest. The RC5 Encryption Algorithm. In *Proc. 1994 Leuven Workshop on Fast Software Encryption*, pages 86–96. Springer-Verlag, 1995.

[27] R. Rivest, M. Robshaw, R. Sidney, and Y. Yin. The RC6™Block Cipher. Specification version 1.1, Aug. 1998.
[28] B. Schneier. *Applied Cryptography: Protocols, Algorithms and Source Code in C*. John Wiley & Sons, Inc., 2nd edition, 1996.
[29] S. Slijepcevic, V. Tsiatsis, S. Zimbeck, M. Srivastava, and M. Potkonjak. On communication security in wireless adhoc sensor networks. In *11th IEEE Int. Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 139–144, June 2002.
[30] Texas Instruments, Inc. MSP430x13x, MSP430x14x Mixed Signal Microcontroller. Datasheet, 2001.
[31] Texas Instruments, Inc. *MSP430x1xx Family User's Guide*, Dec. 2003.
[32] J. Worley, B. Worley, T. Christian, and C. Worley. AES Finalists on PA-RISC and IA-64: Implementations & Performance. In *Proc. 3rd AES Conference (AES3)*, 2001.
[33] W. Ye, J. Heidemann, and D. Estrin. An Energy-Efficient MAC protocol for Wireless Sensor Networks. In *Proc. IEEE Infocom*, pages 1567–1576, New York, NY, USA, June 2002. USC/Information Sciences Institute, IEEE.

Appendix: Per-Cycle Energy Consumption

Different instructions may take different numbers of clock cycles, resulting in different energy consumption per instruction. Even different instructions with the same number of clock cycles may consume different amounts of energy, because of the nature of the instruction itself, for example an instruction that accesses the memory would naturally consume more energy than an instruction that accesses the registers. We will however show that the *energy consumed per cycle* does not vary much from instruction to instruction.

Methodology To achieve this, we should ideally measure the energy consumed by each cycle of different instructions. However measuring such energy is difficult without instrumenting the chip, so we measure the current instead. If we fix the voltage V, by measuring the current I, we get the power P. If a cycle is t_c seconds long, and an instruction consists of c cycles, let $e_1, ..., e_c$ be the energy consumed by each cycle, then

$$VI = P = \frac{e_1 + ... + e_c}{ct_c} = \frac{\bar{e}}{t_c} \implies \bar{e} = VIt_c \quad (3)$$

where $\bar{e} = \frac{e_1+...+e_c}{c}$ is the average energy consumed per cycle. Since V and t_c are fixed, by measuring I, we are in fact measuring $\bar{e}$. There is a possibility that $e_i \ll \bar{e}$ and $e_j \gg \bar{e}$ for some $i \neq j$ and yet $\bar{e}$ does not vary much from instruction to instruction, meaning that even if $\bar{e}$ is constant, we cannot claim that "energy per cycle" is constant. However this is not a problem, because every instruction is always executed as a whole, with the energy-lean cycle(s) compensating the energy-consuming cycle(s). Worth mentioning is that this method is consistent with Chien and Wen's [5].

The current is measured when an instruction `xxx` is executed in an infinite loop:

```
Mainloop    xxx <some randomised operands>
            xxx <other randomised operands>
            ...100 times
            jmp Mainloop
```

Note that in the above template, one `jmp` instruction after every 100 times of the measured instruction does not affect the measurement much, moreover we can measure the current of `jmp` without the influence of other instructions:

```
Mainloop    jmp Label2
Label2      jmp Mainloop
```

Whenever immediate constant operands, offsets, data are involved, they are randomly generated. In fact, all the test programs are generated by a Perl script.

Since there are 7 addressing modes [31], an instruction like `mov.w` can be used in *at least* 7 modes depending on the type of its operands, e.g. '`mov.w R12, 2(R14)`', '`mov.w @R12+, R14`' etc. Fortunately not all modes of the same instruction are generated by the compiler. We only test those modes of the instruction generated by the compiler. To find these in-use modes, we have written a Perl script to parse the assembler code of our block cipher algorithms (generated by the compiler). For example, the only mode used for the instruction `and.b` is '`and.b #C,Rn`', and we only measure this particular mode of `and.b` (where '`#C`' stands for an immediate constant, and '`Rn`' stands for a register).

Our test programs are generally divided into 3 parts: (1) the program, (2) the source data, and (3) the destination data. For example, while the instruction '`mov.w @R12, 2(R14)`' itself resides in the program area, '`@R12`' points to a word in the source data area, whereas '`2(R14)`' points to a word in the destination data area. Referring to Table 11, $I_{\texttt{RAM}}$ refers to the current when the program, source data and destination data are loaded in the RAM; whereas $I_{\texttt{Flash}}$ refers to the current when the program and the source data are loaded in the Flash *but* the destination data in the RAM. The destination data is always loaded in the RAM because they are meant to be overwritten byte-by-byte, while Flash can only be erased one sector at a time. We do not consider cache because there is none in the processor. Logically $I_{\texttt{RAM}}$ is lower than $I_{\texttt{Flash}}$ since accessing the RAM is cheaper, however we will show that the difference is only about 6% of the mean.

Instead of measuring the current consumed by the processor alone, we have measured the current consumed by the entire EYES sensor node. This is acceptable because the measured instructions do not invoke functions on the peripheral circuits, and assuming the leakage current in the peripheral circuits stays constant independent of the measured instructions.

Results There are no entries in Table 11 for '`ret`', '`pop.b Rn`' and '`pop.w Rn`'. Instead, '`ret`' is measured along with '`call #L`' or '`call Rn`'; '`pop.b Rn`' along with '`push.b Rn`'; and '`pop.w Rn`' along with '`push.w #C`', '`push.w Rn`' or '`push.w X(Rn)`'. This is fair because in real-world applications, a `pop` is always associated with a `push`, so is a `ret` associated with a `call`.

The average current is 2.93mA, with a standard deviation of 0.05. From the table, we can see that the most energy-consuming instructions are '`mov.b @Rn+,Rn`' and '`xor.b @Rn+,Rn`', consuming a current of 3.03mA (when the program and the source data are loaded in the Flash), whereas '`bis.w Rn,Rn`', '`mov.b Rn,Rn`', '`mov.w Rn,Rn`', the rotation instructions, '`swpb Rn`' and '`sxt Rn`' are the cheapest, consuming a current of 2.85mA (when everything is loaded in the RAM). While it is easy to appreciate why the latter instructions elicit the least current, it is interesting to learn that the instruction mode of '`@Rn+,Rn`' draws a higher current than '`@Rn+,X(Rn)`' (although this does *not* mean that the '`@Rn+,Rn`' over 2 cycles, consumes more energy than '`@Rn+,X(Rn)`' over 5 cycles).

All in all, the difference between the largest and the smallest current is only 6% of the mean, and we conclude that $\bar{e}$ is more or less consistent, or in other words it is safe to assume that "energy per cycle" is more or less consistent for our particular hardware platform.

Table 11. Measured currents $I_{\texttt{Flash}}$ and $I_{\texttt{RAM}}$ for each instruction in mA (V=2.994V, t_c=0.22μs).

Instruction	c	$I_{\texttt{Flash}}$	$I_{\texttt{RAM}}$
`add.b #C,Rn`	2	2.98	2.89
`add.b Rn,Rn`	1	2.95	2.87
`add.w #C,Rn`	2	2.99	2.91
`add.w #C,X(Rn)`	5	2.99	2.94
`add.w Rn,Rn`	1	2.96	2.87
`addc.w #C,Rn`	1	2.99	2.90
`addc.w #C,X(Rn)`	5	2.97	2.90
`addc.w X(Rn),Rn`	3	2.99	2.90
`addc.w X(Rn),X(Rn)`	6	2.99	2.90
`and.b #C,Rn`	1	2.99	2.89
`and.w #C,Rn`	1	2.99	2.90
`and.w #C,X(Rn)`	5	2.97	2.90
`and.w @Rn,Rn`	2	2.99	2.90
`and.w X(Rn),Rn`	5	3.00	2.90
`bis.w @Rn,Rn`	2	2.98	2.89
`bis.w Rn,Rn`	1	2.93	2.85
`bis.w X(Rn),Rn`	3	2.98	2.89
`bit.b #C,X(Rn)`	5	2.96	2.88
`bit.w #C,Rn`	2	2.98	2.90
`bit.w #C,X(Rn)`	5	2.96	2.89
`br #L`	3	2.96	2.87
`call #L`	5	2.95	2.88
`call Rn`	4	2.96	2.90
`clrc`	1	2.97	2.92
`cmp.w #C,Rn`	2	3.00	2.91
`cmp.w #C,X(Rn)`	5	2.98	2.90
`cmp.w Rn,Rn`	1	2.97	2.87
`cmp.w Rn,X(Rn)`	4	3.00	2.91
`cmp.w X(Rn),Rn`	3	3.00	2.91
`cmp.w X(Rn),X(Rn)`	6	2.99	2.90
`jc L`	2	2.96	2.89
`jeq L`	2	2.96	2.89
`jge L`	2	2.97	2.89
`jl L`	2	2.96	2.89
`jmp L`	2	2.97	2.89
`jnc L`	2	2.97	2.89
`jne L`	2	2.97	2.89
`mov.b #C,Rn`	2	2.98	2.89
`mov.b #C,X(Rn)`	5	2.97	2.89
`mov.b &L,Rn`	3	3.00	2.89
`mov.b @Rn,Rn`	2	3.00	2.89
`mov.b @Rn,X(Rn)`	5	3.00	2.90
`mov.b @Rn+,Rn`	2	3.03	2.91
`mov.b @Rn+,X(Rn)`	5	2.99	2.91
`mov.b Rn,Rn`	1	2.94	2.85
`mov.b Rn,X(Rn)`	4	2.97	2.89
`mov.b X(Rn),Rn`	3	2.99	2.89
`mov.b X(Rn),X(Rn)`	6	2.99	2.89
`mov.w #C,Rn`	2	2.98	2.89
`mov.w #C,X(Rn)`	5	2.98	2.90
`mov.w @Rn,Rn`	2	2.99	2.89
`mov.w @Rn,X(Rn)`	5	2.98	2.90
`mov.w Rn,Rn`	1	2.93	2.85
`mov.w Rn,X(Rn)`	4	2.97	2.89
`mov.w X(Rn),Rn`	3	2.99	2.89
`mov.w X(Rn),X(Rn)`	6	2.98	2.90
`push.b Rn`	3	2.99	2.91
`push.w #C`	4	2.97	2.89
`push.w Rn`	3	3.00	2.91
`push.w X(Rn)`	5	2.98	2.90
`rla.b Rn`	1	2.93	2.85
`rla.w Rn`	1	2.94	2.85
`rlc.b Rn`	1	2.94	2.85
`rlc.w Rn`	1	2.94	2.85
`rra.b Rn`	1	2.93	2.85
`rra.w Rn`	1	2.93	2.85
`rrc.b Rn`	1	2.93	2.85
`rrc.w Rn`	1	2.93	2.85
`sub.b Rn,Rn`	1	2.95	2.86
`sub.w #C,Rn`	2	3.00	2.91
`sub.w @Rn,Rn`	2	3.02	2.89
`sub.w Rn,Rn`	1	2.95	2.87
`sub.w X(Rn),Rn`	3	3.01	2.90
`subc.b #C,Rn`	2	2.99	2.89
`subc.b Rn,Rn`	1	2.95	2.86
`subc.w Rn,Rn`	1	2.95	2.87
`subc.w X(Rn),Rn`	3	3.00	2.91
`swpb Rn`	1	2.94	2.85
`sxt Rn`	1	2.93	2.85
`xor.b #C,Rn`	2	2.99	2.90
`xor.b @Rn,Rn`	2	3.00	2.90
`xor.b @Rn,X(Rn)`	5	2.99	2.91
`xor.b @Rn+,Rn`	2	3.03	2.91
`xor.b Rn,Rn`	1	2.94	2.86
`xor.b Rn,X(Rn)`	4	2.98	2.92
`xor.b X(Rn),Rn`	3	3.00	2.91
`xor.b X(Rn),X(Rn)`	6	2.99	2.90
`xor.w #C,Rn`	2	3.00	2.90
`xor.w #C,X(Rn)`	5	2.99	2.91
`xor.w @Rn,Rn`	2	3.00	2.91
`xor.w Rn,Rn`	1	2.95	2.87
`xor.w Rn,X(Rn)`	4	3.00	2.91
`xor.w X(Rn),Rn`	3	3.00	2.90