Online Appendix to: Survey and Benchmark of Block Ciphers for Wireless Sensor Networks

YEE WEI LAW, JEROEN DOUMEN, and PIETER HARTEL
University of Twente

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.

B. 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 $e_1, \ldots, e_c$ be the energy consumed by each cycle, then

$$VI = P = \frac{e_1 + \cdots + e_c}{ct_c} = \frac{\bar{e}}{t_c} \Longrightarrow \bar{e} = VIt_c \tag{4}$$

where $\bar{e} = \frac{e_1+\cdots+e_c}{c}$ 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 [1998].

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 [Texas Instruments, Inc. 2003], an instruction like `mov.w` can be used in *at least* 7 modes depending on the type of its operands, for example, '`mov.w R12, 2(R14)`', '`mov.w @R12+, R14`' and so on. 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 XI, $I_{\texttt{RAM}}$ 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 we assume the leakage current in the peripheral circuits stays constant independent of the measured instructions.

C. RESULTS

There are no entries in Table XI 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 2.93 mA, 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 3.03 mA (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 2.85 mA (when everything is loaded

Table XI. Measured Currents $I_{\texttt{Flash}}$ and $I_{\texttt{RAM}}$ for Each Instruction in mA ($V = 2.994$ V, $t_c = 0.22$ μs, See Text for the Definition of $I_{\texttt{Flash}}$ and $I_{\texttt{RAM}}$).

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

Instruction	c	$I_{\texttt{Flash}}$	$I_{\texttt{RAM}}$
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

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.