# 轻量级分组密码 RECTANGLE 基于 FELICS 的实现与优化

罗 鹏[1,2], 张文涛[1,3], 包珍珍[1,2]

[1] 中国科学院信息工程研究所信息安全国家重点实验室 北京 中国 100093
[2] 中国科学院大学 北京 中国 100049
[3] 中国科学院大学网络空间安全学院 北京 中国 100049

**摘要** 随着物联网的普及以及 RFID、传感器的广泛应用，轻量级密码算法受到人们越来越多的关注。对于一个轻量级密码算法，除了安全性之外，软件和硬件实现性能也非常重要。卢森堡大学的科研人员于 2015 年开发了一个开源框架——FELICS (Fair Evaluation of Lightweight Cryptographic Systems)，旨在公平地测评轻量级密码算法在嵌入式设备上的软件性能。FELICS 需要在两种应用场景下(一为通信协议，另一为认证协议)，测试一个密码算法在三种嵌入式平台(8 位 AVR、16 位 MSP 以及 32 位 ARM)下运行所需的 Flash、RAM 和执行时间，再对结果取加权平均值，并据此对参赛的轻量级分组密码的软件性能进行综合排名。到目前为止，FELICS 已经包含了 18 个轻量级分组密码。本文首先分析 FELICS 中已提交的分组密码的 C 语言及汇编语言代码，总结常用的优化方法。然后在三种嵌入式平台上实现了轻量级分组密码 RECTANGLE。进一步地，我们对算法轮密钥加、列变换、行移位这三种操作进行了优化。优化后的结果如下：在 ARM 平台，优化后轮函数所需的 Flash 减少 42.6%、同时时间减少 36.8%；在 AVR 平台场景 1 下，优化后 RECTANGLE-128 的 RAM 减少了 12.0%、同时时间减少了 5.0%，RECTANGLE-80 的 RAM 减少了 10.9%、同时时间减少了 2.8%。FELICS 的最终结果显示，在 18 个轻量级分组密码算法中，RECTANGLE 在两种应用场景下分别排名第 4 和第 5 位，这表明 RECTANGLE 在嵌入式平台上具有优秀的软件性能。

**关键词** 轻量级分组密码；RECTANGLE；FELICS；嵌入式设备；软件优化实现
**中图法分类号** TP309.7 DOI 号 10.19363/j.cnki.cn10-1380/tn.2017.07.006

# The Implementation and Optimization of Lightweight Block Cipher RECTANGLE based on FELICS

LUO Peng[1,2], ZHANG Wentao[1,3], BAO Zhenzhen[1,2]

[1] State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China
[2] University of Chinese Academy of Sciences, Beijing 100049, China
[3] School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100049, China

**Abstract** With the popularization of Internet of Things (IOT) and the wide application of RFID and sensors, more attention is being paid to lightweight ciphers. Besides security, the software and hardware performance are also important for a lightweight cipher. The researchers in University of Luxembourg developed an open-source framework——FELICS (Fair Evaluation of Lightweight Cryptographic Systems) in 2015, which aims at fairly evaluating the software performance of lightweight ciphers on embedded devices. By extracting Flash, RAM consumption and execution time on three widely used microcontrollers: 8-bit AVR, 16-bit MSP and 32-bit ARM, the ciphers are ranked respectively with an average value under scenario 1(communication protocol) and scenario 2(authentication protocol). Until now, 18 lightweight block ciphers have entered this competition. In this paper, we firstly analyze the C and assembler implementations, and summarize some common optimization methods. Then, we give the implementations of the lightweight block cipher RECTANGLE on the three devices. Furthermore, we optimize the three basic operations AddRoundKey, SubColumn and ShiftRow. The results are as follows. The Flash reduces 42.6% and the time reduces 36.8% respectively for the round function on ARM. For scenario 1 on AVR, the RAM and time of RECTANGLE-128 reduce 12.0% and 5.0%, with that of RECTANGLE-80 reducing 10.9% and 2.8%. The final results on FELICS show that RECTANGLE ranked 4$^{th}$ under Scenario 1 and 5$^{th}$ under Scenario 2 among the 18 lightweight block ciphers, which indicates that RECTANGLE has very good software performance on embedded devices.

**Key words** lightweight block cipher; RECTANGLE; FELICS; embedded devices; optimized software implementation

---

**通讯作者**：张文涛，博士，副研究员, Email: zhangwentao@iie.ac.cn。
本课题得到国家自然科学基金 (No.61379138), 中国科学院先导专项 (No.XDA06010701)和信息保障技术重点实验室开放基金 (KJ-15-003)资助。
收稿日期：2016-06-26; 修改日期：2016-08-28; 定稿日期：2017-03-06

# 1 FELICS

FELICS[1]是由卢森堡大学的 Daniel 等人在 2015 年开发的一个开源框架，它旨在测评轻量级分组密码和流密码在嵌入式设备(如 AVR，MSP，ARM)上的软件性能。

## 1.1 背景

随着物联网的快速发展，越来越多的嵌入式设备被应用在 RFID、无线传感器、智能卡等网络节点上。预测到 2020 年，物联网将会包括近 500 亿的通信节点[27]，覆盖移动医疗、智能家居、智慧城市等各大领域，因此如何保证网络节点的安全通信也变得非常重要。嵌入式设备的一个显著特征就是资源受限。在软件方面，无论是 RAM、Flash 还是计算能力都比传统的设备要低很多；而在硬件上，嵌入式设备更多地应用在物理空间较小的场景，这要求算法具有很小的实现面积。这些都使得传统的密码算法不能很好地满足要求。近十年来，人们提出了近三十个轻量级分组密码，比如 Chaskey[3]、Fantomas[4]、HIGHT[5]、KLEIN[6]、LBlock[7]、LEA[8]、LED[9]、Piccolo[10]、PRESENT[11]、PRIDE[12]、PRINCE[13]、RECTANGLE[2]、RoadRunneR[14]、Robin[4]、SIMON[15]、SPECK[15]、TWINE[16]，等等。密码算法在提供足够安全性的同时，软硬件性能也是非常重要的指标。但目前没有统一的标准来评价轻量级分组密码算法的软件性能，设计者在设计新算法时也很难准确地与现有算法做公平比较。基于此，Daniel 等人设计了 FELICS, 它在相对公平地比较现有轻量级分组密码算法软件性能的同时, 也为后续的设计者提供了很大的参考价值。

FELICS 具有良好的扩展性，参赛的每一个密码算法需要在 8 位 AVR、16 位 MSP 和 32 位 ARM 上针对两种应用场景实现 4 个接口：RunEncryptionKeySchedule (加密密钥扩展算法，可无)、Encrypt (加密算法)、Decrypt (解密算法)、RunDecryptionKeySchedule (解密密钥扩展算法，可无)，再获取算法运行所需的 Flash、RAM 以及执行时间，最后根据计算的 FOM 值对算法进行排名。FOM 将 3 种平台的结果进行加权平均，用来说明密码算法在嵌入式平台的综合软件性能。其中权重可以有不同的方法，目前 FELICS 的计算方法给 3 种平台的权重相等。

## 1.2 测评场景

场景 0 测试算法加解密一个分组消耗的资源以及运行时间，能反映算法的基本软件性能，但不参与结果排名。

场景 1 用于描述物联网中节点之间的安全通信。根据 IEEE 802.15.4 和 ZigBee 标准中对传感网络之间通信协议的描述，在单一事务中，物联网设备之间通信的数据为 128 字节。因此场景 1 假定通信数据固定为 128 字节，使用轻量级分组密码在 CBC 模式下加解密。

场景 2 用于描述物联网设备之间消息的认证，通信双方需要交互的数据固定为 128 比特，并在 CTR 模式下工作。由于只是认证，因此不需要解密算法；而轮子密钥固定存储在 Flash 中，故也不需要密钥扩展算法。

## 1.3 测评平台

FELICS 测试平台分别是 8 位 AVR ATmega128[24]、16 位 MSP430F1611[25]和 32 位 Atmel SAM3X8 Cortex M3[21]。

AVR 微处理器共 133 条指令，AVR 支持单操作数和双操作数指令，大多数指令占用 2 字节，基本的算术指令、逻辑指令、移位指令只需要 1 个时钟周期即可完成。条件跳转指令需要的时钟周期一般比无条件跳转指令少，但跳转范围有限，一般为[-64B, 64B)，而无条件跳转可以达到几 MB。AVR 共有 32 个 8 位寄存器(r0-r31)。其中，6 个寄存器能被用来作为 16 位的寻址地址，分别是 X(r27:r26)、Y(r29:r28)、Z(r31:r30)。X、Y、Z 都能被用来作为访问 RAM 的地址，但如果要访问 Flash，则只能通过 Z。另外，乘法的结果只能保存在(r1:r0)中，立即数操作指令要求寄存器下标不小于 16(即 r16-r31)。AVR 提供了一条特殊的赋值指令(MOVW)，它可以在 1 个时钟周期内完成 16 比特的赋值，但要求源操作寄存器和目的寄存器下标均为偶数。

MSP430 共 51 条指令，包括 27 条基本指令和 24 条模拟指令 (通过基本指令模拟实现)。基本指令分为 7 条单操作数指令、8 条转移指令和 12 条双操作数指令。其中大多数指令提供面向 16 比特字和 8 比特字节两种类型的操作，默认情况下都是面向字的操作，可以通过在指令操作符后面加上.*b* 将指令限定为字节操作，字节操作指令会将目的寄存器的高 8 位清 0。MSP430 一共拥有 16 个寄存器(r0-r15)，其中 4 个特殊寄存器为程序计数器(r0 或 pc)、栈指针(r1 或 sp)、状态寄存器(r2 或 sr/cg1)以及常数生成器(r3 或 cg2)，剩余 12 个为通用寄存器。与一般微处理器指令不同的是，MSP430 的双操作数指令中，第一个是源操作数，第二个是目的操作数。

ARM Cortex M3 很好地支持三操作数指令。一条指令一般占用 2 或 4 字节，对于常用的加减算术

指令、逻辑指令、移位指令，1个时钟周期即可完成。其中循环右移指令一次可以移多比特，而循环左移和循环右移具有等价性，因此没有循环左移指令。ARM提供的比特域操作、字节/字扩展等指令可以为优化带来很大的空间。此外，ARM的访存指令非常丰富，轮密钥加的实现部分会对此做详细介绍。ARM处理器拥有16个寄存器(r0-r15)，r0-r12为通用寄存器，r13-r15为特殊寄存器：其中r13是栈指针(sp)；r14 是链接寄存器(lr)，用来保存函数调用的返回地址；r15是程序计数器(pc)。

### 1.4 测评指标

密码算法在每种平台上的实现，FELICS关注的指标有三项：Flash、RAM 和执行时间。Flash 包括代码和数据两部分，RAM包括栈消耗和数据消耗。数据部分相对固定，通常包括轮常量、S盒、初始向量、轮子密钥、明文等，而代码量、栈消耗、执行时间则来自函数 RunEncryptionKeySchedule、RunDecryptionKeySchedule、Encrypt、Decrypt的运行。

FELICS分别对密码算法在4种编译条件下进行测试，分别是O1、O2、O3和Os，不同的选项反映了编译器针对性能、代码量不同的优化强度。文献[28]中给出了相关描述。其中O1提供最基本的优化，包括程序中的自增/自减(auto-inc-dec)、分支延迟(delayed-branch)等，O2、O3比O1、O2的优化强度更大，而Os则在O2基础上专门针对代码量进行优化。

## 2 优化方法及已有结果分析

FELICS 目前的结果包括 AES[17]、Chaskey、Chaskey-LTS、Fantomas、HIGHT、LBlock、LEA、LED、Piccolo、PRESENT、PRIDE、PRINCE、RECTANGLE、RC5[18]、RoadRunneR、Robin、SIMON、SPECK、TWINE。本节将分析这些密码算法在实现过程常用的优化方法，并对一些算法的性能做总结。

### 2.1 优化方法

参加 FELICS 竞赛的密码算法常用的优化方法有查表、减少访存次数、循环展开、利用三操作数指令等。实际实现中，为了达到最优的结果，往往都是多种优化方法联合使用。

1) 查表

查表的思想是通过空间换取时间。查表技术常用的地方是将S盒、线性层等多步的最终结果保存，进而减少中间计算需要的代码和时间。例如AES_128_128_v06版本将算法S盒、行移位、列混淆合并为一次查表操作，通过最终的结果可以看到，对于场景1的AVR平台，同样在O1的编译选项下，AES_128_128_v06的执行时间(102658 cycles)比不采用这种技术的AES_128_128_v01(531876 cycles)快了5倍多。但查表技术无法并行处理，一种用于实现S盒的并行技术叫比特切片[22]。

2) 减少访存次数

对于任何CPU，访存指令（包括从RAM访问数据和从Flash访问数据）都是时间消耗相对较高的，减少访存次数可以在一定程度上提高性能。例如MSP和ARM，一次可以尽可能多地将数据加载到寄存器中，再在寄存器中提取所需要的数据，而寄存器操作比访存操作所需要的时间少很多。但对于AVR，每次访存只能加载 8 位数据，减少访存的方法不适用。

3) 循环展开

循环展开的思想是通过空间换取时间，其本质是减少循环控制语句的执行次数、减少单轮操作可能存在的数据转移，但代价是轮函数代码近似成倍增加。比如Simon_64_128_v05和Simon_64_128_v06就分别对轮函数进行了2轮、4轮的展开，从而消除了一轮操作所需要的两条赋值操作。

4) 三操作数指令

利用 ARM 三操作数指令可以减少指令的执行时间。例如 $a = b + c$ 操作，在AVR上需要2条指令(一条加法ADD，一条赋值MOV)来实现，但ARM一条ADD指令即可完成。

### 2.2 已有结果分析

通过对算法的实现代码以及最终结果进行分析，可以得出如下结论：

1) SIMON、SPECK在3种平台的性能都很好

SIMON、SPECK 64位的分组长度被分成32位的两部分操作。在AVR上可以用4个寄存器表示、MSP可以用2个寄存器表示、ARM可以直接用一个寄存器表示，使得寄存器的价值得到了充分利用。同时两个算法的操作非常简单，只涉及循环移位、异或、按位与操作，即使是在MSP上寄存器也完全够用，因此总体性能非常好。

2) AES总体性能一般，在AVR上性能很好

AES在3种平台都给出了汇编实现，同时针对各平台的特点作了大量的优化，这都使得AES的结果较好。AES的最小操作单元是字节，可以与AVR的一个寄存器对应，而AVR拥有丰富的寄存器，因此AES在AVR上的实现性能很好。

3) PRESENT在3种平台的性能都不佳

PRESENT 的16个S盒一般通过查表实现 (并行实现代价非常大)，S盒的输入是 4 比特，需要额

外的指令进行提取。另外，PRESENT 的线性层主要面向硬件设计，在硬件中可以直接通过拉线实现，但在软件实现中需要对每一比特单独操作。这些导致 PRESENT 的软件性能不好，三个平台的结果均排在了后面。

4) LBlock 性能比 PRESENT 优

LBlock 的整体结构类似于 Feistel 网络，在轮函数部分使用了 SPN 网络。32 位输入经过 8 个不同的 4 比特 S 盒，需要额外的指令提取出 S 盒输入。相比 PRESENT，LBlock 的线性层直接对 S 盒的输出做整体交换，因此软件实现上比 PRESENT 稍快。

### 2.3 存在的不足

FELICS 为密码算法软件性能的公平比较提供了很好的平台，经过近一年的发展已经获得了很大关注，但仍有一些需要改进、完善的地方。

1) 同等条件下，分组长度为 128 位的算法比 64 位算法在时间测试上稍有优势：FELICS 为了保持统一，限定了密码算法的加解密函数只能实现一个分组的加解密，对于多个分组则通过外部函数循环调用。由于加密数据量相同，对于分组长度为 128 位的算法，Encrypt、Decrypt 的调用次数只有 64 位算法的一半，因此函数的调用与返回、寄存器入栈与出栈、程序的初始化等执行时间少了一半。

2) C 语言实现灵活简单；汇编与具体指令结合更紧密，能够使用的优化技巧相对较多，但目前很多算法没有对应的汇编实现。相信随着时间的推移以及更多密码学工作者的关注，这将会得到很大改善。

## 3 RECTANGLE 基于 FELICS 实现与优化

说明，本文用到的一些符号说明如下：

a) $LSB_n(R)$ 表示取 $R$ 最低 n 比特(最右边 n 比特)；$MSB_n(R)$ 表示取 $R$ 最高 n 比特(最左边 n 比特)；

b) $\|$ 表示二进制串的拼接；

c) 表格中指令用到 S7-S0、T3-T0、P 分别为对应平台的寄存器，下标值越大，表示的位数越高。例如若[S7，S6，S5，S4，S3，S2，S1，S0]表示算法在 AVR 的状态，则 S7 是最高 8 位，S0 是最低 8 位；若[S3, S2, S1, S0]表示算法在 ARM 的状态，则 $LSB_{16}(S3)$ 为最高 16 位，$LSB_{16}(S0)$ 为最低 16 位。

### 3.1 算法介绍

RECTANGLE[2]是由张文涛等人设计的轻量级分组密码。RECTANGLE 的整体结构为 SPN 网络，分组长度为 64 比特，密钥长度分为 80 比特和 128 比特两个版本(下文分别用 RECTANGLE-80 和 RECTANGLE-128 表示)。本小节对算法做简要介绍，详细内容请参考文献[2]。

#### 3.1.1 状态

**定义 1.** 64 比特明文、密文、轮子密钥以及中间结果统称为算法的一个状态。

64 比特状态 $W = w_{63} \| w_{62} \| \cdots \| w_2 \| w_1 \| w_0$ 统一按照图 1 排列为 $4\times16$ 的矩阵。图 2 为对应的二维矩阵表示方法，其中：

a) $R_i = a_{i,15} \| \cdots \| a_{i,2} \| a_{i,1} \| a_{i,0}, 0 \leqslant i \leqslant 3$，表示算法状态的一行；

b) $C_j = a_{3,j} \| a_{2,j} \| a_{1,j} \| a_{0,j}, 0 \leqslant j \leqslant 15$，表示算法状态的一列。

$$\begin{bmatrix} w_{15} & \cdots & w_2 & w_1 & w_0 \\ w_{31} & \cdots & w_{18} & w_{17} & w_{16} \\ w_{47} & \cdots & w_{34} & w_{33} & w_{32} \\ w_{63} & \cdots & w_{50} & w_{49} & w_{48} \end{bmatrix}$$

**图 1 算法状态**

$$\begin{array}{ccccc} \left[\begin{array}{ccccc} a_{0,15} & \cdots & a_{0,2} & a_{0,1} & a_{0,0} \\ a_{1,15} & \cdots & a_{1,2} & a_{1,1} & a_{1,0} \\ a_{2,15} & \cdots & a_{2,2} & a_{2,1} & a_{2,0} \\ a_{3,15} & \cdots & a_{3,2} & a_{3,1} & a_{3,0} \end{array}\right] \begin{array}{l} \to R_0 \\ \to R_1 \\ \to R_2 \\ \to R_3 \end{array} \\ \begin{array}{ccccc} \downarrow & & \downarrow & \downarrow & \downarrow \\ C_{15} & & C_2 & C_1 & C_0 \end{array} \end{array}$$

图 2 二维坐标表示

#### 3.1.2 轮函数

RECTANGLE 一共有 25 轮，每轮包括轮密钥加、列变换、行移位 3 步。最后增加一次轮密钥加操作。

1) **轮密钥加：**64 比特的中间状态与轮密钥按位异或；

2) **列变换：**对每一列进行 S 盒替换，即 $S(C_j) = a'_{3,j} \| a'_{2,j} \| a'_{1,j} \| a'_{0,j}$，其中 $0 \leqslant j \leqslant 15$、$C_j$ 为 4 比特输入；

3) **行移位：**对每一行进行左循环移位，即 $R'_0 = R_0 <<< 0$、$R'_1 = R_1 <<< 1$、$R'_2 = R_2 <<< 12$、$R'_3 = R_3 <<< 13$，其中 $<<<$ 表示 16 比特字的循环左移。

#### 3.1.3 密钥扩展算法

- RECTANGLE-80：80 位密钥 $V = v_{79} \| \cdots \| v_2 \| v_1 \| v_0$ 排列成 $5\times16$ 的矩阵。$R_i = v_{i\times16+15} \| \ldots \| v_{i\times16}$，用来表示第 $i$ 行，其中 $0 \leqslant i \leqslant 4$。第一轮密钥 $K_0 = R_3 \| R_2 \| R_1 \| R_0$，接着循环执行如下 3 步，每循环一次产生一个轮子密钥，一共循环 25 次。

1) S 盒操作：对最低 4 列进行 S 盒替换，即 $v'_{3,j} \| v'_{2,j} \| v'_{1,j} \| v'_{0,j} = S(v_{3,j} \| v_{2,j} \| v_{1,j} \| v_{0,j})$，其中 $0 \leqslant j \leqslant 3$；

2) 一轮 5 分支广义 Feistel 变换：$R'_0 = (R_0 <<< 8) \oplus R_1$，$R'_1 = R_2$，$R'_2 = R_3$，$R'_3 = (R_3 <<< 12) \oplus R_4$，$R'_4 = R_0$；

3) 轮常量异或：$v'_{0,4} \| v'_{0,3} \| v'_{0,2} \| v'_{0,1} \| v'_{0,0} = (v_{0,4} \| v_{0,3} \| v_{0,2} \| v_{0,1} \| v_{0,0}) \oplus RC[i]$，其中 $0 \leqslant i \leqslant 24$，$RC[i]$ 表示第 $i$ 轮的常量。

● RECTANGLE-128：128 比特的主密钥 $V = v_{127} \| v_{126} \| \cdots \| v_2 \| v_1 \| v_0$ 排列成 $4 \times 32$ 的矩阵，$R_i = v_{i \times 32+31} \| \ldots \| v_{i \times 32}$，用来表示第 $i$ 行，其中 $0 \leqslant i \leqslant 3$。则第一轮子密钥 $K_0 = LSB_{16}(R_3) \| LSB_{16}(R_2) \| LSB_{16}(R_1) \| LSB_{16}(R_0)$。接着循环执行如下步骤 25 次。

1) S 盒操作：对最低 8 列进行 S 盒替换，即 $v'_{3,j} \| v'_{2,j} \| v'_{1,j} \| v'_{0,j} = S(v_{3,j} \| v_{2,j} \| v_{1,j} \| v_{0,j})$，$0 \leqslant j \leqslant 7$；

2) 一轮 4 分支广义 Feistel 变换：$R'_0 = (R_0 <<< 8) \oplus R_1$，$R'_1 = R_2$，$R'_2 = (R_2 <<< 16) \oplus R_3$，$R'_3 = R_0$；

3) 轮常量异或：和 RECTANGLE-80 相同。

## 3.2 实现与优化

文献[19]、[20]和[21]分别给出了 AVR、MSP 和 ARM 的指令集。本节主要介绍 RECTANGLE 轮密钥加、列变换、行移位在 3 种平台的实现与优化。

### 3.2.1 轮密钥加

AVR 一次访存只能操作 8 位，64 位的轮子密钥需要 8 次访存和异或操作。AVR 有两类指令分别用来访问 RAM 和 Flash，其中 LD (ST)用于 RAM 的读(写)，LPM (SPM)用于 Flash 读 (写)。一条 LD (ST)指令需要 2 cycles，一条 LPM (SPM)需要 3 cycles。

MSP 一次访存可以操作 16 位数据，需要 4 次访存和异或操作来实现一次轮密钥加。MSP 没有独立的访存指令，无论是 RAM 还是 Flash，都通过 MOV 指令实现数据的读写。MSP 异或指令(XOR)可以从 RAM(或 Flash)加载数据并与寄存器异或，相当于将 MOV 和 XOR 一起进行，可以减少 Flash。例如 MSP 中“MOV @X+, T”和“XOR T, S0”两条指令可以实现第 0 行的轮密钥加，这一共需要 4 字节，3 cycles；但通过“XOR @X+, S0”实现第 0 行的轮密钥加则只需要 2 字节，2 cycles。

表 1 是 ARM 实现轮密钥加的不同方法，每种方法的第 1 列表示指令，第 2、3 列分别表示对应指令消耗的 Flash 和时间。ARM 的访存指令支持字节(LDRB，STRB)、半字 (LDRH，STRH)、字 (LDR，STR)、双字 (LDRD，STRD) 以及多寄存器的访存指令 (LDM，STM)。Method 1 中每次加载 16 位轮密钥，与状态中的一行求异或，一共需要 24 bytes、12 cycles。Method 2 利用 LDR 指令一次加载 32 位数据以及第二个操作数可以使用循环移位的特点，Flash 减少 4 bytes，执行时间减少 4 cycles。Method 3 利用 LDM 指令一次实现全部 64 位轮密钥的加载，同时利用了 EORS 指令在源操作数与目的操作数相同的情况下只占用 2 字节的特点，进一步减少了 6 bytes 和 1 cycle。最终实现一次轮密钥加只需要 14 bytes、7 cycles。Flash 和时间都减少 41.7%。

**表 1 轮密钥加在 ARM 上的优化**

| Method 1 | | | Method 2 | | | Method 3 | | |
|---|---|---|---|---|---|---|---|---|
| Instruction / Flash(bytes) / Time(cycles) | | | | | | | | |
| LDRH T0, [P, #0] | 2 | 2 | **LDR T0, [P, #0]** | 2 | 2 | **LDM P!, {T0, T1}** | 2 | 3 |
| EOR S0, S0, T0 | 4 | 1 | EOR S0, S0, T0 | 4 | 1 | **EORS** S0, S0, T0 | 2 | 1 |
| LDRH T0, [P, #2] | 2 | 2 | EOR S1, S1, **T0, LSR #16** | 4 | 1 | EOR S1, S1, **T0, LSR #16** | 4 | 1 |
| EOR S1, S1, T0 | 4 | 1 | **LDR T0, [P, #4]** | 2 | 2 | **EORS** S2, S2, T1 | 2 | 1 |
| LDRH T0, [P, #4] | 2 | 2 | EOR S2, S2, T0 | 4 | 1 | EOR S3, S3, **T1, LSR #16** | 4 | 1 |
| EOR S2, S2, T0 | 4 | 1 | EOR S3, S3, **T0, LSR #16** | 4 | 1 | | | |
| LDRH T0, [P, #6] | 2 | 2 | | | | | | |
| EOR S3, S3, T0 | 4 | 1 | | | | | | |

### 3.2.2 列变换

RECTANGLE 的 S 层基于比特切片方法设计，S 层在实现时通过逻辑运算 (赋值、异或、逻辑或、按位与、按位取反)可以并行实现多个 S 盒替换。

表 2 为 S 盒在三种平台的指令。RECTANGLE 在密钥扩展、加密、解密部分分别需要用到 S 盒。

在密钥扩展算法中，RECTANGLE-128 密钥状态最低 8 列需要经过 S 盒，对于 AVR 可以直接操作相应寄存器，但 MSP 和 ARM 需要额外的指令保存与恢复其余不变的比特位。RECTANGLE-80 只有低 4

**表 2　S 盒在 AVR、MSP 和 ARM 上的实现**

| AVR S 盒 | | | MSP S 盒 | | | ARM S 盒 | | |
|---|---|---|---|---|---|---|---|---|
| Instruction / Flash(bytes) / Time(cycles) | | | | | | | | |
| MOVW T0, S4 | 2 | 1 | MOV S2, T0 | 2 | 1 | ORN T0, S3, S1 | 4 | 1 |
| EOR S4, S2 \| EOR S5, S3 | 2×2 | 2×1 | XOR S1, S2 | 2 | 1 | EOR(S) T0, T0, S0 | 2 | 1 |
| COM S2 \| COM S3 | 2×2 | 2×1 | INV S1 | 2 | 2 | BIC(S) S0, S0, S1 | 2 | 1 |
| MOVW T2, S0 | 2 | 1 | MOV S0, T1 | 2 | 1 | EOR T1, S2, S3 | 4 | 1 |
| AND S0, S2 \| AND S1, S3 | 2×2 | 2×1 | AND S1, S0 | 2 | 1 | EOR(S) S0, S0, T1 | 2 | 1 |
| OR S2, S6 \| OR S3, S7 | 2×2 | 2×1 | BIS S3, S1 | 2 | 1 | EOR S3, S1, S2 | 4 | 1 |
| EOR S6, T0 \| EOR S7, T1 | 2×2 | 2×1 | XOR T1, S1 | 2 | 1 | EOR S1, S2, T0 | 4 | 1 |
| EOR S0, S6 \| EOR S1, S7 | 2×2 | 2×1 | XOR T0, S3 | 2 | 1 | AND(S) T1, T0, T1 | 2 | 1 |
| EOR S2, T2 \| EOR S3, T3 | 2×2 | 2×1 | XOR S3, S0 | 2 | 1 | EOR(S) S3, S3, T1 | 2 | 1 |
| AND S6, S2 \| AND S7, S3 | 2×2 | 2×1 | AND S1, S3 | 2 | 1 | ORR S2, S0, S3 | 4 | 1 |
| EOR S6, S4 \| EOR S7, S5 | 2×2 | 2×1 | XOR S2, S3 | 2 | 1 | EOR(S) S2, S2, T0 | 2 | 1 |
| OR S4, S0 \| OR S5, S1 | 2×2 | 2×1 | BIS S0, S2 | 2 | 1 | | | |
| EOR S4, S2 \| EOR S5, S3 | 2×2 | 2×1 | XOR S1, S2 | 2 | 1 | | | |
| EOR S2, T0 \| EOR S3, T1 | 2×2 | 2×1 | XOR T0, S1 | 2 | 1 | | | |

列经过 S 盒，因此 3 个平台都需要额外的指令保存不变的数据。

对于加解密，16 列都需要经过 S 盒替换，AVR 中除了 MOVW 可以操作 16 位数据，所有的逻辑运算只能操作 8 位，因此 AVR 在加解密部分 S 盒的实现中，异或 (EOR)、逻辑或 (OR)、按位与 (AND) 和按位取反 (COM)相应地都需要两条指令。

MSP 和 ARM 一条指令可以完成 16 位的操作，同时，ARM 利用三操作数指令的优势减少了指令条数。ARM 平台逻辑指令一般占用 4 字节，但当目的寄存器与一个源寄存器相同时，可以利用 Thumb-2 指令减少 Flash 消耗。例如“EOR T0, T0, S0”占用 4 字节，但“EORS T0, T0, S0”只占用 2 字节；修改后执行时间没有变化。最终列变换在 ARM 上实现所需的 Flash 可以从 44 字节变为 32 字节，降低 27.3%。

### 3.2.3　行移位

**定义 2.**　ROR_X_N 表示将 N 比特数据循环右移 X 位，ROL_X_N 表示将 N 比特数据循环左移 X 位，其中$1 \leqslant X \leqslant N-1$。

根据定义可知：ROR_X_N 等价于 ROL_(N-X)_N。加密用到的操作有：ROL_1_16，ROL_12_16 和 ROL_13_16；解密用到的操作有：ROR_1_16，ROR_12_16 和 ROR_13_16；RECTANGLE 密钥扩展算法用到的循环移位操作有：ROL_8_16，ROL_12_16，ROL_8_32 和 ROL_16_32。ARM 上可以一次移位多位。AVR 和 MSP 只有 1 位的循环移位指令，只能逐步实现多位循环移位。

**表 3　AVR 循环移位的实现**

| ROL_1_16 | | ROL_3_16 | | ROL_4_16 | | ROR_1_16 | | ROR_3_16 | | ROR_4_16 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Instruction / Flash(bytes) / Time(cycles) | | | | | | | | | | | |
| LSL S0 | 2 1 | ROL_1_16 | 6 3 | SWAP S0 | 2 1 | BST S0, 0 | 2 1 | ROR_4_16 | 14 7 | SWAP S0 | 2 1 |
| ROL S1 | 2 1 | ROL_1_16 | 6 3 | SWAP S1 | 2 1 | ROR S1 | 2 1 | ROL_1_16 | 6 3 | SWAP S1 | 2 1 |
| ADC S0, zero | 2 1 | ROL_1_16 | 6 3 | MOV T0, S0 | 2 1 | ROR S0 | 2 1 | | | MOV T0, S0 | 2 1 |
| | | | | EOR T0, S1 | 2 1 | BLD S1, 7 | 2 1 | | | EOR T0, S1 | 2 1 |
| | | | | ANDI T0, 0X0F | 2 1 | | | | | ANDI T0, 0XF0 | 2 1 |
| | | | | EOR S0, T0 | 2 1 | | | | | EOR S0, T0 | 2 1 |
| | | | | EOR S1, T0 | 2 1 | | | | | EOR S1, T0 | 2 1 |

注: a) zero 为一个值为 0 的寄存器

1) AVR

对于 16 位的字，循环左移(或右移)的范围是[0, 15]，根据定义 2，加解密的 6 种循环移位操作可以等价为 ROL_1_16、ROR_4_16、ROR_3_16、ROR_1_16、ROL_4_16、ROL_3_16。表 3 给出了几种循环移位的实现。

ROL_8_16、ROL_8_32 和 ROL_16_32 只在密钥扩展算法部分使用，主密钥在 AVR 中的状态如图 3、4 所示，其中 $k0$-$k15$ 均为一个字节。表 4 给出了密钥扩展算法中行变换在 AVR 上的实现指令。MSP 和 ARM 平台密钥扩展的实现思路与 AVR 一致，此处不再具体介绍。

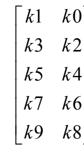


**图 3　80 位密钥状态**

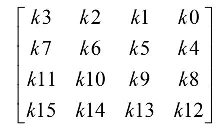


**图 4　128 位密钥状态**

**表 4　密钥扩展算法中行变换在 AVR 上实现**

| RECTANGLE-80 | | | | RECTANGLE-128 | | | |
|---|---|---|---|---|---|---|---|
| Instruction / Flash(bytes) / Time(cycles) | | | 说明 | Instruction / Flash(bytes) / Time(cycles) | | | 说明 |
| MOVW t0, k8 | 2 | 1 | T = R4 | MOVW t0, k12 | 2 | 1 | T = R3 |
| MOVW k8, k0 | 2 | 1 | **R4′ = R0** | MOVW t2, k14 | 2 | 1 | |
| MOVW k0, k2 | 2 | 1 | R0′ = R1 | MOVW k12, k0 | 2 | 1 | **R3′ = R0** |
| MOVW k2, k4 | 2 | 1 | **R1′ = R2** | MOVW k14, k2 | 2 | 1 | |
| MOVW k4, k6 | 2 | 1 | **R2′ = R3** | MOVW k0, k4 | 2 | 1 | R0′ = R1 |
| MOVW k6, t0 | 2 | 1 | R3′ = R4 | MOVW k2, k6 | 2 | 1 | |
| EOR k0, k9 | 2 | 1 | **R0′ = (R0<<<8) ⊕ R1** | MOVW k4, k8 | 2 | 1 | **R1′ = R2** |
| EOR k1, k8 | 2 | 1 | | MOVW k6, k10 | 2 | 1 | |
| MOVW t0, k4 | 2 | 1 | T = R3 | MOVW k8, t0 | 2 | 1 | R2′ = R3 |
| SWAP t0 | 2 | 1 | T = R3<<<12 | MOVW k10, t2 | 2 | 1 | |
| SWAP t1 | 2 | 1 | | EOR k0, k15 | 2 | 1 | **R0′ = (R0<<<8) ⊕ R1** |
| MOV t2, t0 | 2 | 1 | | EOR k1, k1 | 2 | 1 | |
| EOR t2, t1 | 2 | 1 | | EOR k2, k13 | 2 | 1 | |
| ANDI t2, 0XF0 | 2 | 1 | | EOR k3, k14 | 2 | 1 | |
| EOR t0, t2 | 2 | 1 | | EOR k8, k6 | 2 | 1 | **R2′ = (R2<<<16) ⊕ R3** |
| EOR t1, t2 | 2 | 1 | | EOR k9, k7 | 2 | 1 | |
| EOR k6, t0 | 2 | 1 | **R3′ = (R3<<<12) ⊕ R4** | EOR k10, k4 | 2 | 1 | |
| EOR k7, t1 | 2 | 1 | | EOR k11, k5 | 2 | 1 | |

2) MSP

表 5 给出了各种循环移位在 MSP 的指令序列。

其中各基本指令 Flash 均为 2 字节，运行时间需要 1 cycle。ROL_1_16 通过逻辑左移和加法实现，需要 4 字节、2 cycles；ROR_1_16 通过右移和比特指令实现，需要 4 字节、2 cycles。对于 ROL_3_16、ROL_4_16、ROR_3_16 和 ROR_4_16，则可以分别通过多次循环左移或右移一位来实现。

3) ARM

ARM 循环移位指令可以移多位，表 6 给出了 ROL_1_16 的实现及优化过程，其中 32 位寄存器的低 16 位为有效位。其他循环移位操作原理类似。最终行移位在 ARM 平台上 Flash 降低 55.6%，时间降低 60.0%。

ROL_8_32 和 ROL_16_32 等价为 ROR_24_32 和 ROR_16_32，通过一条 ROR 指令实现。

### 3.2.4　AVR 上另一种优化实现

图 5、6 表示 RECTANGLE-80、RECTANGLE-128 一轮密钥扩展的示意图，其中 $S_i(X)$ 表示 $X$ 的每一比特都是 S 盒输入的第 $i$ 比特。从图中可以看出，RECTTANGLE-80 中上一轮的子密钥 $k5$ 、$k7$ 仍出现在下一轮中(RECTANGLE-128 中上一轮的 $k1$ 和 $k9$ 出现在下一轮)，因此除了第一轮需要保存完整的 8 字节密钥外，后续每一轮只需要保存 6 字节，这样

**表 5　MSP 基本循环移位的实现**

| ROL_1_16 | | | ROL_3_16 | | | ROL_4_16 | | | ROR_1_16 | | | ROR_3_16 | | | ROR_4_16 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Instruction / Flash(bytes) / Time(cycles) | | | | | | | | | | | | | | | | | |
| RLA S0 | 2 | 1 | ROL_1_16 | 4 | 2 | ROL_1_16 | 4 | 2 | BIT #1, S0 | 2 | 1 | ROR_1_16 | 4 | 2 | ROR_1_16 | 4 | 2 |
| ADC S0 | 2 | 1 | ROL_1_16 | 4 | 2 | ROL_1_16 | 4 | 2 | RRC S0 | 2 | 1 | ROR_1_16 | 4 | 2 | ROR_1_16 | 4 | 2 |
| | | | ROL_1_16 | 4 | 2 | ROL_1_16 | 4 | 2 | | | | ROR_1_16 | 4 | 2 | ROR_1_16 | 4 | 2 |
| | | | | | | ROL_1_16 | 4 | 2 | | | | | | | ROR_1_16 | 4 | 2 |

**表 6 ROL_1_16 在 ARM 上的实现及优化**

| Method 1 | | | Method 2 | | |
|---|---|---|---|---|---|
| Instruction / Flash(bytes) / Time(cycles) | | | | | |
| AND S0,S0,#0X0000FFFF | 4 | 1 | **BFI S0, S0, #16, #15** | 4 | 1 |
| LSL S0, S0, #1 | 4 | 1 | ROR S0, S0, #15 | 4 | 1 |
| MOV T0, S0 | 2 | 1 | | | |
| LSR T0, T0, #16 | 4 | 1 | | | |
| EOR S0, S0, T0 | 4 | 1 | | | |

$$\begin{bmatrix} k1 & k0 \\ k3 & k2 \\ k5 & k4 \\ k7 & k6 \\ k9 & k8 \end{bmatrix} \rightarrow \begin{bmatrix} (MSB_4(k0) \| S_0(LSB_4(k0))) \oplus k3 & k1 \oplus (MSB_4(k2) \| S_1(LSB_4(k2))) \oplus RC[i] \\ k5 & MSB_4(k4) \| S_2(LSB_4(k4)) \\ k7 & MSB_4(k6) \| S_3(LSB_4(k6)) \\ (S_3(LSB_4(k6)) \| MSB_4(k7)) \oplus k9 & (LSB_4(k7) \| MSB_4(k6)) \oplus k8 \\ k1 & MSB_4(k0) \| S_0(LSB_4(k_0)) \end{bmatrix}$$

**图 5 80 位密钥扩展算法**

$$\begin{bmatrix} k3 & k2 & k1 & k0 \\ k7 & k6 & k5 & k4 \\ k11 & k10 & k9 & k8 \\ k15 & k14 & k13 & k12 \end{bmatrix} \rightarrow \begin{bmatrix} k2 \oplus k7 & k1 \oplus k6 & S_0(k0) \oplus k5 & k3 \oplus S_1(k4) \oplus RC[i] \\ k11 & k10 & k9 & S_2(k8) \\ k9 \oplus k15 & S_2(k8) \oplus k14 & k11 \oplus k13 & k10 \oplus S_3(k12) \\ k3 & k2 & k1 & S_0(k0) \end{bmatrix}$$

**图 6 128 位密钥扩展算法**

一共可以减少 50 字节的轮子密钥。在加解密部分，通过两个额外的寄存器保存上一轮不变的两字节子密钥，进行轮密钥加时每一轮只需要 6 次访存操作，进一步能够减少执行时间。

表 7 给出了优化 RAM 前后，在 AVR 平台下场景 1 的最优结果，其中 RECTANGLE-80 的 RAM 降低了 10.9%，时间减少了 2.8%；RECTANGLE-128 的 RAM 降低了 12.0%，时间减少了 5.0%。

**表 7 优化 RAM 前后 AVR 的最优结果**

| | 优化 RAM 前 | | | 优化 RAM 后 | | |
|---|---|---|---|---|---|---|
| 算法 | Flash(bytes) | RAM(bytes) | Time(cycles) | Flash(bytes) | RAM(bytes) | Time(cycles) |
| RECTANGLE-80 | 1122 | 395 | 68630 | 1152 | 352 | 66722 |
| RECTANGLE-128 | 1108 | 401 | 68249 | 1118 | 353 | 64813 |

## 4 RECTANGLE 的实现结果以及与其他算法比较

表 8 给出了 RECTANGLE-80、RECTANGLE-128 在场景 1、场景 2 下各平台综合性能最优的结果。综合性能是指 FOM 最小的，但并不表示在某一项指标上最优。比如 AES 一般查表的综合性能最优，但查表会消耗更多的 Flash；如果不采用查表方式实现，则消耗的 Flash 会减少，但同时执行时间会增加。

附录一给出了 RECTANGLE-128 各个版本在场景 1 下的详细结果，RECTNAGLE-80 以及其他算法的详细结果请参阅 FELICS 公布的数据[26]。

### 4.1 RECTANGLE 与其他算法结果的比较

到目前为止，总共有 18 个轻量级分组密码参赛。在场景 1 下，RECTANGLE 排名第 4，前 3 名分别是 Chaskey、SPECK、SIMON；在场景 2 下 RECTANGLE 排名第 5，前 4 名分别是 Chaskey、SPECK、SIMON、LEA。

表 9 给出了 Chaskey、SIMON、SPECK、RECTANGLE、AES、PRESENT 等算法在场景 1 下各平台的最优结果。从表 9 中可以看出，对于 AES，在 AVR 平台，RECTANGLE 的执行时间稍差，但 FLASH 减少 60%多；在 MSP 平台，RECTANGLE 的 Flash 和执行时间分别减少 68%和 46%；在 ARM 平台，RECTANGLE 的 Flash 和执行时间分别减少近 78%和 50%。相比 ISO 现有轻量级分组密码标准 PRESENT，RECTANGLE 在 3 种平台的优势非常明显，特别在执行时间上提升了好几倍。在 AVR 和 MSP 平台，RECTANGLE 的性能与 SIMON 基本相当甚至优于 SIMON；在 ARM 上比 SIMON 稍差一些。

**表 8　RECTANGLE 综合性能最优结果**

| 密码算法信息 | 场景 | AVR | | | MSP | | | ARM | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Flash(bytes) / RAM(bytes) / Time(cycles) | | | | | | | | |
| RECTANGLE-80 | 1 | 1152 | 352 | 66722 | 818 | 396 | 45688 | 670 | 426 | 36814 |
| RECTANGLE-128 | 1 | 1118 | 353 | 64813 | 844 | 402 | 46196 | 654 | 432 | 37006 |
| RECTANGLE-80 | 2 | 602 | 56 | 4381 | 480 | 54 | 2651 | 452 | 76 | 2432 |
| RECTANGLE-128 | 2 | 606 | 56 | 4433 | 480 | 54 | 2651 | 452 | 76 | 2432 |

**表 9　FELICS 场景 1 下密码算法的最终结果**

| 密码算法信息 | | | AVR | | | MSP | | | ARM | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 算法 | 分组长度(比特) | 密钥长度(比特) | Flash(bytes) / RAM(bytes) / Time(cycles) | | | | | | | | |
| Chaskey | 128 | 128 | 1510 | 229 | 22142 | 1136 | 244 | 23402 | 438 | 236 | 9851 |
| SPECK | 64 | 96 | 966 | 294 | 39875 | 556 | 288 | 31360 | 492 | 308 | 15427 |
| SPECK | 64 | 128 | 874 | 302 | 44895 | 572 | 296 | 32333 | 444 | 308 | 16505 |
| Chaskey-LTS | 128 | 128 | 1510 | 229 | 34814 | 1140 | 244 | 37626 | 438 | 236 | 12859 |
| SIMON | 64 | 96 | 1084 | 363 | 63649 | 738 | 360 | 47767 | 600 | 376 | 23056 |
| SIMON | 64 | 128 | 1122 | 375 | 66613 | 760 | 372 | 49829 | 560 | 392 | 23930 |
| RECTANGLE | 64 | 80 | 1152 | 352 | 66722 | 818 | 396 | 45688 | 670 | 426 | 36814 |
| RECTANGLE | 64 | 128 | 1118 | 353 | 64813 | 844 | 402 | 46196 | 654 | 432 | 37006 |
| AES | 128 | 128 | 3010 | 408 | 58246 | 2684 | 408 | 86506 | 3050 | 452 | 73868 |
| PRESENT | 64 | 80 | 2160 | 448 | 245232 | 1818 | 448 | 202050 | 2116 | 470 | 274463 |

### 4.2　RECTANGLE 与 ATmega128 上 AES 硬件加密比较

ATmega128 提供了 AES 硬件加密功能，MICAz 是一款基于 ATmega128 处理器的芯片，Healy[29]在 MICAz 芯片上给出了 AES 硬件加密的时间，如表 10 所示。其中轮子密钥固定存储在 Flash，INIT 包括将轮子密钥、明文写入 RAM 以及从 RAM 读取密文的时间；ENC 是加密 16 字节的时间。

**表 10　ATmega128 平台 AES 硬件加密**

| 算法 \ 内容 | INIT (us) | ENC (us) |
|---|---|---|
| AES | 294.003 | 29.831 |

**表 11　ATmega128 平台 RECTANGLE 软件加密**

| 算法 \ 内容 | KS(cycles) | ENC (cycles) |
|---|---|---|
| RECTANGLE-80 | 1772 | 1849 |
| RECTANGLE-128 | 1399 | 1823 |

表 11 给出了 RECTANGLE 在 ATmega128 平台密钥扩展(KS)、加密一个分组(ENC)的时间，该数据根据 FELICS 场景 0 结果得到。由于 RECTANGLE 一个分组是 8 字节，因此加密 16 字节所需的总时间可以通过 Equ. 1)计算，其中 $F$ 是 ATmega128 处理器的频率，值为 16MHz。

$$T = \frac{KS + ENC * 2}{F} \qquad \text{Equ. 1)}$$

最终的时间如表 12 所示，从中可以看出，RECTANGLE 在 ATmaga128 上的汇编实现和 AES 硬件实现的加密速度相当，这进一步说明 RECTANGLE 具备非常好的软件性能。

**表 12　ATmega128 上 AES 硬件加密与 RECTANGLE 软件加密比较**

| 算法 \ 内容 | 总时间 (us) |
|---|---|
| AES | 323.83 |
| RECTANGLE-80 | 341.88 |
| RECTANGLE-128 | 315.31 |

## 5　未来工作

密码算法软件性能的提升可以从各个密码组件的分析优化入手，组件的实现得到了优化，则密码算法自身的软件性能将得到提升。本文基于三种微处理器平台，分别从轮密钥、列变换和行移位实现并优化 RECTANGLE。对于轮密钥加和行移位，主要结合具体平台的指令集特点，挖掘可能存在的高效实现；对于列变换，则主要通过 Gladman[23]给出的 S 盒指令序列搜索程序进行搜索，后续我们将加

入具体平台的指令特点，尝试寻找 S 盒更优的指令序列。同时，对于其他应用较广的平台，我们将考虑给出 RECTANGLE 相应的实现。

## 参考文献


[1] D. Dinu, A. Biryukov, J. Großschädl, D. Khovratovich, Y.L. Corre and L. Perrin, “FELICS–Fair Evaluation of Lightweight Cryptographic Systems,” in *NIST Workshop on Lightweight Cryptography*, 2015.

[2] W.T. Zhang, Z.Z Bao, D.D Lin, V. Rijmen, B.H. Yang and I. Verbauwhede, “RECTANGLE: a bit-slice lightweight block cipher suitable for multiple platforms,” *Science China Information Sciences*, vol. 58, no. 12, pp. 1-15, Dec. 2015.

[3] N. Mouha, B. Mennink, A.V Herrewege, D. Watanabe, B. Preneel and I. Verbauwhede, “Chaskey: an efficient MAC algorithm for 32-bit microcontrollers,” in Proc. *Selected Areas in Cryptography (SAC’14)*, pp. 306-323, 2014.

[4] V. Grosso, G. Leurent, F.X. Standaert and K. Varıcı, “LS-designs: Bitslice encryption for efficient masked software implementations,” in Proc. *Fast Software Encryption (FSE’14)*, pp. 18-37, 2014.

[5] D. Hong, J. Sung, S. Hong, J. Lim, S. Lee, B.S. Koo, C. Lee, D. Chang, J. Lee, K. Jeong, H. Kim, J. Kim and S. Chee, “HIGHT: A new block cipher suitable for low-resource device,” in Proc. *Cryptographic Hardware and Embedded Systems (CHES’06)*, pp. 46-59, 2006.

[6] Z. Gong, S. Nikova and Y.W. Law, “KLEIN: a new family of lightweight block ciphers,” in Proc. *RFID Security and Privacy (RFIDSec’11)*, pp. 1-18, 2011.

[7] W.L. Wu, L. Zhang, “LBlock: a lightweight block cipher,” in Proc. *Applied Cryptography and Network Security (ACNS’11)*, pp. 327-344, 2011.

[8] D. Hong, J.K. Lee, D.C. Kim, D. Kwon, K.H. Ryu and D.G. Lee, “LEA: A 128-bit block cipher for fast encryption on common processors,” in Proc. *Information Security Applications (ISA’13)*, pp. 3-27, 2013.

[9] J. Guo, T. Peyrin, A. Poschmann and M. Robshaw, “The LED block cipher,” in Proc. *Cryptographic Hardware and Embedded Systems (CHES’11)*, pp. 326-341, 2011.

[10] K. Shibutani, T. Isobe, H. Hiwatari, A. Mistuda, T. Akishita and T. Shirai, “Piccolo: An Ultra-Lightweight Blockcipher,” in Proc. *Cryptographic Hardware and Embedded Systems (CHES’11)*, pp. 342-357, 2011.

[11] A. Bogdanov, L.R. Knudsen, G. Leander, C. Paar, A. Poschmann, M.J.B. Robshaw, Y. Seurin and C. Vikkelsoe, “PRESENT: An ultra-lightweight block cipher,” in Proc. *Cryptographic Hardware and Embedded Systems (CHES’07)*, pp. 450-466, 2007.

[12] M.R. Albrecht, B. Driessen, E.B. Kavun, G. Leander, C. Paar, T. Yalçın, “Block ciphers–focus on the linear layer (feat. PRIDE),” in Proc. *Advances in Cryptology (CRYPTO’14)*, pp. 57-76, 2014.

[13] J. Borghoff, A. Canteaut, T. Güneysu, E.B. Kavun, M. Knežević, L.R. Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger, P. Rombouts, S.S. Thomsen and T. Yalçın, “PRINCE–a low-latency block cipher for pervasive computing applications,” in Proc. *Advances in Cryptology (ASIACRYPT’12)*, pp. 208-225, 2012.

[14] A. Baysal and S. Şahin, “Roadrunner: A small and fast bitslice block cipher for low cost 8-bit processors,” in Proc. *Lightweight Cryptography for Security and Privacy (LCSP’15)*, pp. 58-76, 2015.

[15] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks and L. Wingers, “The SIMON and SPECK lightweight block ciphers,” in Proc. *Design Automation Conference, 2015 52nd ACM/EDAC/IEEE*, pp.1-6, 2015.

[16] T. Suzaki, K, Minematsu, S. Morioka and E. Kobayashi, “Twine: A lightweight, versatile block cipher,” in Proc. *ECRYPT Workshop on Lightweight Cryptography (ECRYPT’11)*, pp. 146-169, 2011.

[17] J. Daemen and V. Rijmen, “AES Proposal: Rijndael”, The Advanced Encryption Standard, 2002.

[18] R.L. Rivest, “The RC5 encryption algorithm,” in Proc. *Fast Software Encryption (FSE’94)*, pp. 86-96, 1994.

[19] “Atmel AVR 8-bit Instruction Set”, http://www.atmel.com/images/atmel-0856-avr-instruction-set-manual.pdf, May. 2016.

[20] “MSP430 Quick Reference”, http://www.ece.utep.edu/courses/web3376/Links_files/MSP430 Quick Reference.pdf, May. 2016.

[21] “Atmel. ARM Cortex-M3 Revision r2p0 Technical Reference Manual”, http://infocenter.arm.com/help/topic/com.arm.doc.ddi0337h /CHDDIGAC.html, May. 2016.

[22] D.A. Osvik, “Speeding up Serpent,” in Proc. *AES Candidate Conference*, pp. 317-329, 2000.

[23] “Software for efficient boolean function decompositions for the eight Serpent S boxes and their inverses,” B.R. Gladman, http://www.gladman.me.uk/, May. 2016.

[24] “Atmel. AVR ATmega128 datasheet”, http://www.atmel.com/images/doc2467.pdf, May. 2016.

[25] “Texas Instruments. MSP430F1611 datasheet”, http://www.ti.com/lit/ds/symlink/msp430f1611.pdf, May. 2016.

[26] “FELICS Block Ciphers Detailed Results,” D. Dinu, https://www.Cryptolux.org/index.php/FELICS_Block_Ciphers_Detailed_Resuls, May. 2016.

[27] D. Evans, “The internet of things: How the next evolution of the internet is changing everything”, CISCO white paper, 1, 1-11, 2011.

[28] “Options That Control Optimization”, https://gcc.gnu.org/ onlinedocs/gcc/Optimize-Options.html, May. 2016.

[29] M. Healy, T. Newe, E. Lewis, “Analysis of Hardware Encryption Versus Software Encryption on Wireless Sensor Network Motes”, in Proc. *Smart Sensors and Sensing Technology (SSST’08)*, pp. 3-14, 2008.

# 附录

表13给出了RECTANGLE-128各版本在场景1下的结果。其中Flash、RAM和执行时间分别表示密钥扩展算法、加密算法和解密算法总的值。

**表13 RECTANGLE-128各个版本结果**

| 版本 | 编译选项 | AVR | | | MSP | | | ARM | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Flash(bytes) | RAM(bytes) | Time(cycles) | Flash(bytes) | RAM(bytes) | Time(cycles) | Flash(bytes) | RAM(bytes) | Time(cycles) |
| RECTANGLE_64_128_v02 | -O1 | 1132 | 401 | 69140 | 900 | 404 | 49073 | 772 | 448 | 40153 |
| | -O2 | 1108 | 401 | 68249 | 880 | 402 | 49574 | 768 | 440 | 38887 |
| | -O3 | 1788 | 411 | 68870 | 1292 | 408 | 46467 | 1180 | 472 | 36097 |
| | -Os | 1110 | 399 | 70042 | 882 | 400 | 49762 | 736 | 440 | 40933 |
| RECTANGLE_64_128_v03 | -O1 | 1132 | 401 | 69140 | 900 | 404 | 49073 | 792 | 436 | 48877 |
| | -O2 | 1108 | 401 | 68249 | 880 | 402 | 49574 | 828 | 424 | 43232 |
| | -O3 | 17 88 | 411 | 68870 | 1292 | 408 | 46467 | 1240 | 456 | 41302 |
| | -Os | 1110 | 399 | 70042 | 882 | 400 | 49762 | 728 | 428 | 48900 |
| RECTANGLE_64_128_v04 | -O1 | 1580 | 406 | 104960 | 990 | 400 | 67701 | 792 | 436 | 48877 |
| | -O2 | 1466 | 404 | 100879 | 982 | 402 | 68404 | 828 | 424 | 43232 |
| | -O3 | 2142 | 414 | 101482 | 1518 | 408 | 65122 | 1240 | 456 | 41302 |
| | -Os | 1582 | 402 | 116766 | 968 | 396 | 68412 | 728 | 428 | 48900 |
| RECTANGLE_64_128_v05 | -O1 | 1142 | 353 | 65704 | 1378 | 364 | 94773 | 1000 | 390 | 57927 |
| | -O2 | 1118 | 353 | 64813 | 1316 | 362 | 93874 | 1040 | 386 | 63331 |
| | -O3 | 1798 | 363 | 65434 | 1852 | 368 | 90592 | 1456 | 418 | 60180 |
| | -Os | 1120 | 351 | 66606 | 1298 | 360 | 98910 | 1004 | 398 | 65587 |
| RECTANGLE_64_128_v06 | -O1 | 1132 | 401 | 69140 | 864 | 404 | 45695 | 692 | 440 | 38593 |
| | -O2 | 1108 | 401 | 68249 | 844 | 402 | 461 96 | 688 | 432 | 37327 |
| | -O3 | 1788 | 411 | 68870 | 1256 | 408 | 43089 | 1100 | 464 | 34537 |
| | -Os | 1110 | 399 | 70042 | 846 | 400 | 46384 | 654 | 432 | 37006 |

**罗鹏** 于 2014 年在武汉理工大学计算机科学与技术专业获得学士学位。现在中国科学院信息工程研究所信息安全专业攻读硕士学位。研究领域为对称密码算法的分析、实现与优化。Email：luopengxq@gmail.com

**包珍珍** 于 2010 年在中国科学技术大学信息安全专业获得学士学位。现在中国科学院信息工程研究所信息安全专业硕博连读。研究领域为对称密码算法的设计、分析以及软件高效实现。Email：baozhenzhen@iie.ac.cn

**张文涛** 于 2004 年在中国科学院软件研究所获得博士学位。现为中国科学院信息工程研究所第一研究室副研究员。研究领域为对称密码算法的设计与分析。Email：zhangwentao@iie.ac.cn