

文章编号:2095-6134(2015)06-0816-09

# 轻量级分组密码 RECTANGLE 在 X86 和 X64 平台的软件实现评估\*

公丽丽<sup>†</sup>, 张文涛, 包珍珍, 郭 淳

(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 100093)

(2014 年 12 月 2 日收稿; 2015 年 3 月 30 日收修改稿)

Gong L L, Zhang W T, Bao Z Z, et al. Evaluation of software implementation of lightweight block cipher RECTANGLE on X86 and X64 platforms[J]. Journal of University of Chinese Academy of Sciences, 2015,32(6):816-824.

**摘 要** 轻量级密码是当前密码学研究的一个热门课题,设计硬件实现和软件实现性能均衡的轻量级密码算法已成为趋势. 尽管在轻量级密码算法软件实现方面已经有一些理论和实际的研究,但是公平地比较不同算法的软件实现性能仍然很困难. 切片实现是软件实现时模仿硬件实现的方式. 本文给出 RECTANGLE 在 X86 和 X64 平台上切片实现以及速度测试结果. 结果表明,RECTANGLE 在中高端平台展现了非常优秀的软件实现性能. 在 2.9 GHz Intel (core) i5-4570s CPU 平台 RECTANGLE 单块加解密速度分别为 34.2 和 30.9 cycles/byte;使用 SSE 指令集,加解密速度分别为 5.2 和 5.1 cycles/byte;使用 AVX 指令集,加解密速度分别为 2.6 和 2.5 cycles/byte.

**关键词** 轻量级密码; 软件实现; RECTANGLE; 切片实现

中图分类号:TN918.4 文献标志码:A doi:10.7523/j.issn.2095-6134.2015.06.014

## Evaluation of software implementation of lightweight block cipher RECTANGLE on X86 and X64 platforms

GONG Lili, ZHANG Wentao, BAO Zhenzhen, GUO Chun

(State Key Laboratory of Information Security, Institute of Information and Engineering, Chinese Academy of Sciences, Beijing 100093, China)

**Abstract** Lightweight cipher has attracted much attention from the cipher community. Nowadays it is a trend to design lightweight cipher which performs well in both hardware and software. Although several theoretical and practical studies have been reported in the field of software implementation of lightweight cipher, it is still difficult to make a fair comparison of software implementation among different lightweight ciphers. In this paper we firstly present an approach of bitslice implementation of RECTANGLE, and then give its evaluation of software performance on X64 and X86 platforms. On 2.9 GHZ Intel (core) i5-4570s CPU, RECTANGLE achieves 34.2 cycles/byte for encryption

\* 国家自然科学基金(61379138)和中国科学院先导专项(XDA06010701)资助

<sup>†</sup> 通信作者, E-mail: gonglili8@163.com

and 30.9 cycles/byte for decryption for one block; on SSE instruction it achieves 5.2 cycles/byte for encryption and 5.1 cycles/byte for decryption; and on AVX instruction it achieves 2.6 cycles/byte for encryption and 2.5 cycles/byte for decryption.

**Key words** lightweight cipher; software implementation; RECTANGLE; bitslice implementation

轻量级密码是当前密码学研究的一个热门课题,这与普适计算的不断发展密切相关。普适计算的特点是大量部署传感节点设备,比如 RFID 标签或无线传感节点,这些设备的计算能力、电量及存储都十分有限,同时大量部署导致成本成为重要约束。普适计算广泛应用于手机、智能卡、动物和货物追踪、电子护照、支付、工业数据采集与控制等方面,在这些应用中必须保护敏感信息的安全性和私密性。传统的密码算法比如 AES 目前最小的实现面积为 2 400 GE (Gate Equivalence),吞吐量为 57 Kbps (处理器主频 100 KHz 时)<sup>[1]</sup>,过低的吞吐量导致较高的数据处理延时与能耗<sup>[2]</sup>,很难满足受限设备在硬件实现面积、能耗等方面的限制。为计算资源受限的设备所设计的存储空间小、运行能耗低、安全性适中的密码算法称为轻量级密码算法<sup>[3-6]</sup>,比如作为 ISO/IEC 轻量级分组密码标准的 PRESENT<sup>[7]</sup>实现面积为 1 570 GE,吞吐量为 200 Kbps (处理器主频 100 KHz 时)。

计算资源受限设备大体分为两类,一类使用固定线路或小型集成电路完成计算任务,如智能卡、专用集成电路等;第二类拥有自己的微处理器、存储空间等,体系结构与微机相似,如传感器节点等。目前已提出很多轻量级密码算法,包括分组密码、流密码、密码杂凑函数、消息认证码等<sup>[7-12]</sup>。早期轻量级密码应用环境设定为计算资源受限设备,第一类受限设备的功能逻辑往往被实现为硬件线路嵌入到硬件设备中,最初这些算法都是面向硬件设计<sup>[3,4]</sup>,目标是减少硬件实现的资源消耗,包括硬件实现面积、数据处理时的能耗等。尽管硬件实现中 GE 是衡量算法的一个重要指标,但跟现实应用关联性有限<sup>[13]</sup>。

2007 年 CHES 会议上 Bogdanov 等<sup>[7]</sup>提出的 PRESENT 以其简洁的设计、引人注目的硬件实现性能及高度的安全性引人关注,并在 2012 年被确定为 ISO/IEC 轻量级密码标准。随后涌现了一批优秀的轻量级密码算法如 KATAN/KATANAN<sup>[9]</sup>、KLEIN<sup>[14]</sup>、LBLOCK<sup>[15]</sup>、TWINE<sup>[16]</sup>、SIMON/SPECK<sup>[17]</sup>及 RECTANGLE<sup>[18]</sup>、LS-Designs<sup>[19]</sup>等。

随着轻量级密码的发展,人们逐渐意识到,软件实现也应该是一个重要的设计准则。第一,在一些低端处理器应用场景比如第二类受限设备,实际上要大量运行分组密码的软件实现实例,专用的硬件实现所占的比例相对小一些<sup>[13]</sup>。第二,轻量级密码算法对中高端平台的应用也非常具有吸引力。在实际应用中,受限设备需跟其他受限设备或服务器交互,一个服务器可能需要跟若干设备进行交互,密码算法可能仅应用在协议中保证交互的安全性,而服务器可能还需运行许多其他应用,用于密码算法方面的计算资源有限,轻量级密码是很好的选择<sup>[6]</sup>。另外在物联网中,分析大数据已经成为竞争的核心,云计算在分析大数据中具有重要作用。在云计算中如果使用轻量级密码算法实现数据的私密性保护,在传感节点加密节点数据时该算法的硬件实现性能很好,而在云端解密时该算法的软件实现性能很好<sup>[5]</sup>,这将是一种非常具有竞争力的解决方案。第三,软件实现比硬件实现更加灵活、成本更低。只要实现平台支持,软件实现的成本可以达到最低。若需修改算法,软件实现更加容易。

与 AES 等传统分组密码算法相比,轻量级密码算法在软件实现方面的数据十分匮乏<sup>[5-6,20-21]</sup>,即使是已经成为 ISO/IEC 轻量级密码标准的 PRESENT<sup>[7]</sup>,也仅有 4、8、16、32 比特处理器的软件实现测试数据。虽然轻量级密码算法在硬件实现上的测试数据比较丰富<sup>[3-4]</sup>,但硬件实现和软件实现的衡量指标不同。很多运算在硬件实现上的代价非常小,但在软件实现上很困难,比如缺乏规律性的比特置换。硬件实现代价小的算法可能软件实现性能并不好。即使是一些面向软件设计的轻量级密码,由于设计时通常考虑适用于低端平台,多数将降低存储作为软件实现时最关注的因素,导致在高端平台上某些密码算法的软件实现性能并不好<sup>[5]</sup>。

本文将给出 RECTANGLE<sup>[18]</sup>在 X86 和 X64 平台的切片实现方法和测试结果。

# 1 轻量级密码算法软件实现

## 1.1 测试平台

我们选择 X86、X64 两种架构的 CPU 作为实验平台,它们是目前中高端机器的主流,同时切片实现在中高端机器上的优势更为明显<sup>[5-6]</sup>.

X86 平台是指一类以 32 位数据作为处理单元的通用计算机,处理器的通用寄存器长度为 32 bit,是 X64 先前的版本.但现在 X64 平台的普及,多数 X86 平台测试环境是在 64 位机器上运行 32 位模式实现的,本文中所用到的 X86 平台也是通过这种方式搭建的.与 X64 架构相比,X86 架构有 8 个 32 比特通用寄存器,8 个 128 比特 SSE 寄存器,寄存器数量比较少.

X64 平台是指一类 64 位数据作为处理单元的通用计算机,处理器的通用寄存器长度为 64 bit.X64 机器主宰了台式电脑、笔记本、上网本和各种服务器领域,现在售出的大多数台式机、笔记本和服务器都是 X64 架构.X64 架构是一种向后兼容的扩展 X86 架构,包括 AMD 64 和 Intel 64.X64 架构拥有 16 个 64 比特通用寄存器、16 个 128 比特 SSE 寄存器,目前最新处理器具有 16 个 256 比特的 YMM 寄存器.X64 平台支持的指令集包括 X86 指令集及其 64 位扩展指令集(比如作用在 32 位和 64 位数据上的 AND、XOR 和 ROL 等逻辑运算指令),以及 128 位和 256 位功能扩展指令集(比如 SSE、SSE2、SSSE、SSE4、AVX 和 AVX2 等),程序可以自由地使用这些指令.特别地,X64 平台与 X86 平台相比,除可寻址的内存范围更大外,它更好地支持了 64 位数据和 64 位向量数据上的操作,比如 64 位数据上的比特操作 LZCNT、POPCNT 和 BEXR 等,更好地提升整数运算的功能、范围和并行性.

## 1.2 测试方法

密码算法的软件实现评估主要分为对软件实现速度的测试和存储空间的测试.在计算资源十分受限的设备上实现时需评估存储空间,比如 ATiny45 微控制器<sup>[21]</sup>仅有 4 KB 闪存作为指令存储空间,256 B SRAM 和 256 B EEPROM 作为数据存储空间;ATmega163 微控制器<sup>[13]</sup>的存储空间为 16 KB 闪存、512 B EEPROM 和 1 024 B SRAM;RL78<sup>[22]</sup>微控制器的 ROM 大小在 2 KB 到 512 KB 之间,RAM 大小在 256 B 到 32 KB 之间.评估存

储通常包括代码存储需求和特殊数据的存储,在上述微控制器平台,代码的存储一般在闪存或是 ROM 中.如果 RAM 大小足够,特殊数据包括算法中 S 盒的存储存放在 RAM 中,否则存放在 ROM 中.为节省存储空间,密钥编排算法一般同加密算法同时进行,因此子密钥一般存在 RAM 中.密码算法的实现速度通常通过测试实现时间来衡量.

在中高端平台和低端平台上,轻量级密码算法软件实现关注的指标是不同的.在低端平台上由于 ROM、RAM 大小及能耗的限制,更为关注存储空间需求以及功耗<sup>[14,22]</sup>.但在中高端平台上,通常内存可达 2 G、4 G 甚至更多,一级缓存可达 16 KB、32 KB,二级缓存可达 256 KB、512 KB 甚至更多<sup>[6]</sup>.本文给出的 RECTANGLE<sup>[18]</sup>切片实现方式需存储 25 轮轮密钥(200 bytes),每轮需 4 个 XMM/YMM 寄存器存储密码状态、8 个 XMM/YMM 寄存器作为临时寄存器、1 个 XMM/YMM 寄存器存储轮密钥,因此这里不考虑存储空间,只关注软件实现速度.

通常使用的测试时间的机制有两种,一种是 X86 系列 RDTSC 指令,另一种使用 C 语言 clock() 函数<sup>[23]</sup>.RDTSC 只在 X86 下有效,其余的平台有类似指令来做准确计数,RDTSC 指令的精度是可以达到 1 ns.C 语言的 clock() 函数是 C/C++ 中的计时函数,对于短时间的定时或延时可以达到 ms 级别,为保证计时精度,可能消耗很长时间.两种计时方式相比,RDTSC 机制计时精度更高同时方便省时但仅限于包含此机制的平台,而 clock() 函数应用平台更广,由于文中测试平台为 X86 和 X64,因此选择 RDTSC 机制.

## 1.3 切片实现

分组密码软件实现方法目前主要有 3 种,查表实现、切片实现、内部指令实现.在这 3 种方法中,查表实现是一种很简单的实现技巧,但是需要的存储一般比较大,同时容易受到时间攻击和 cache 攻击等侧信道攻击威胁.查表实现中可以通过 PSHUFB 指令实现查表,但该指令依赖于处理器支持的指令集,在受限设备中一般不支持这样的指令集.内部指令实现方式虽然可以使算法的软件实现达到很大优化,但目前在分组密码中只有 AES 算法被 Intel 做成了内部指令,AES128 加解密速度为 1.38 cycles/byte<sup>[24]</sup>.

切片技术实现密码算法是 Biham 在 1997 年

引入以提升 DES 软件实现的性能<sup>[25]</sup>. 在 1990 后期,DES 的切片实现用在 DES 的密钥暴力破解中. 切片的基本思想是软件实现中模拟硬件实现方式. 将整个算法表示成一系列逻辑运算,在一个带有  $n$  比特寄存器的处理器中,执行一条逻辑指令相当于同时执行  $n$  个硬件逻辑门. 在切片实现中,S 盒通过比特的逻辑运算实现,而不是查表,因为这些指令的执行时间跟输入和密钥的值无关,所以切片实现是抵抗时间攻击和 Cache 攻击的.

切片实现技术自提出之后,一直受众人关注,并得到了很好的发展. Biham<sup>[25]</sup> 提出的分组密码的切片实现,是在 64 比特处理器上加密或解密 64 个 DES 分组. Matsui 和 Nakajima<sup>[26]</sup> 证实 Intel core2 处理器上充分利用 SIMD 架构可以达到优越的性能提升. 而实现过程中的一个难点是同时处理  $n$  个不同的分组, Konighofer<sup>[27]</sup> 给出在 64 比特平台上每次并行处理 4 个分组的方式. Kasper 和 Schwabe<sup>[28]</sup> 在此基础上扩展在 Core2 上 AES 切片实现,实现速度达到加密 7.59 cycles/byte. Seiichi 和 Shiho<sup>[5]</sup> 给出了 PRESENT 和 LED 的切片实现. Ryad 等<sup>[6]</sup> 给出切片实现中输入输出时转换为切片实现格式所占的时间以及查表实现、使用 VPERM 技术实现、切片实现 3 种方式适用的场景. Serpent<sup>[29]</sup> (AES 征集时最后的 5 个候选算法之一)、NOEKEON<sup>[30]</sup> (Nessie 候选算法之一)、Keccak<sup>[31]</sup> (SHA-3) 以及 JH<sup>[32]</sup> (SHA-3 征集时最后的 5 个候选算法之一) 等密码算法,都利用切片实现提升其软件实现速度.

2 RECTANGLE 算法

RECTANGLE<sup>[18]</sup> 是迭代分组密码,分组长度是 64 bit,密钥长度是 80 或 128 bit.

2.1 密码状态和子密钥状态

将 64 bit 明文分组、64 bit 中间状态、64 bit 密文状态统称为密码状态. 在加密和解密过程中,密码状态可以表示为一个  $4 \times 16$  的矩形比特阵列. 假设 64 bit 的密码状态记为  $W = w_{63} \parallel \cdots \parallel w_1 \parallel w_0$ , 比特矩阵如图 1 所示. 与密码状态一样,80 或 128 bit 密钥也可用  $4 \times 20$  或  $4 \times 32$  的比特矩阵来表示. 为了方便描述,以下密码状态用二维表示(见图 1).

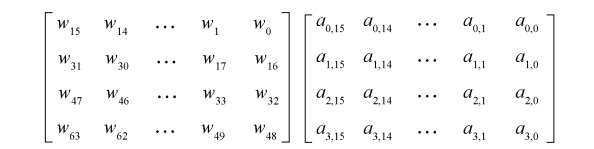


图 1 密码状态及二维表示  
Fig. 1 Cipher state and two-dimensional representation of RECTANGLE

2.2 轮函数

RECTANGLE 是一个 25 轮的 SPN 结构的分组密码. 每一轮包括以下 3 步: 轮密钥加 AddRoundKey, 列替换 SubColumn, 行移位 ShiftRow. 25 轮后,增加一个轮密钥加.

轮密钥加:将轮密钥异或到中间状态上.

列替换:并行执行 16 个相同的 S 盒. 同一列 4 个比特作为一个 S 盒的输入,如图 2(a). 使用的 S 盒是 4 bit 到 4 bit 的非线性置换,即  $S: F_2^4 \rightarrow F_2^4$ . S 盒如表 1 所示.

表 1 RECTANGLE 的 S 盒  
Table 1 S box of RECTANGLE

$x$	0	1	2	3	4	5	6	7
$S(x)$	9	4	F	A	E	1	0	6
$x$	8	9	A	B	C	D	E	F
$S(x)$	C	7	3	8	2	B	5	D

行移位:对每一行左循环移位不同的参数,用  $\lll(x)$  循环左移  $x$  位,如图 2(b).

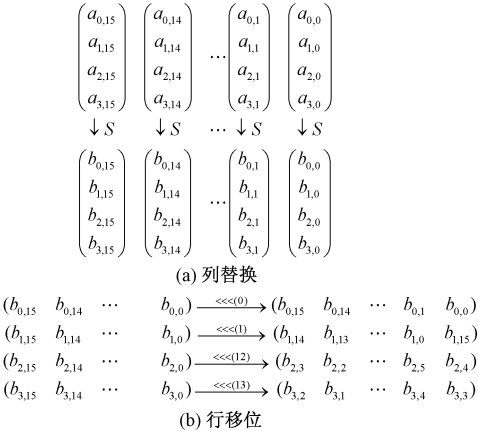


图 2 比特矩阵中的各列执行列替换和行移位操作  
Fig. 2 SubColumn and ShiftRow operates on the columns of the state

2.3 密钥编排算法

下面给出 80 bit 密钥编排算法. 对于用户提供的 80 bit 密钥  $V = v_{79} \parallel \cdots \parallel v_0$ , 首先将其表示



为  $4 \times 20$  的比特矩阵形式, 比特矩阵与图 1 类似, 轮密钥的二维表示记为  $V = (k_{3,19} \parallel \cdots \parallel k_{3,0}) \parallel \cdots \parallel (k_{0,19} \parallel \cdots \parallel k_{0,0})$ .

第  $i(i = 0, 1, \cdots, 24)$  轮的 64 bit 的轮密钥由密钥状态每行最右边的 16 bit 组成, 即  $K_i = (k_{3,15} \parallel \cdots \parallel k_{3,0}) \parallel \cdots \parallel (k_{0,15} \parallel \cdots \parallel k_{0,0})$ . 抽取轮密钥后密钥状态按照以下方式更新:

- 1) 第 0 列执行 S 盒替换,  
 $k_{3,0} \parallel k_{2,0} \parallel k_{1,0} \parallel k_{0,0} : = S(k_{3,0} \parallel k_{2,0} \parallel k_{1,0} \parallel k_{0,0})$ .
- 2) 行移位: 每行左循环移位不同的参数, 第 0 行到第 3 行分别移位 7, 9, 11, 13 bit.
- 3) 异或常量:  
 $k_{0,4} \parallel k_{0,3} \parallel k_{0,2} \parallel k_{0,1} \parallel k_{0,0} : = (k_{0,4} \parallel k_{0,3} \parallel k_{0,2} \parallel k_{0,1} \parallel k_{0,0}) \oplus RC[i]$ . 常量见设计文献[18].

3 Rectangle 的软件实现测试结果

RECTANGLE<sup>[18]</sup> 算法基于切片实现而设计, 使用切片实现方式具有很大的优势.

3.1 RECTANGLE 的 3 种软件实现方法

我们将以下面 3 种方式实现 RECTANGLE: 使用通用指令集、使用 SSE 指令集和使用 AVX 指令集. 密钥编排同加解密算法分开, 时间测试也是分开的. 每次测试时, 先测试密钥编排所用时间并将轮子密钥存储, 再分别测试在 CTR 模式下加密和解密不同分组数时的时间.

密钥编排部分, 对 80 bit 和 128 bit 的单个主密钥进行密钥扩展, 经过扩展得到各轮 64 bit 的轮子密钥. 以 80 bit 密钥编排说明实现方法, 先将主密钥载入 4 个 32 bit 变量 (每个变量载入 20 bit), 抽取 64 bit 作为轮密钥, 密钥状态更新时 S 盒部分使用跟加密过程相同的指令序列<sup>[18]</sup>, 行移位部分使用 SHIFT 和 XOR 指令.

对加解密部分, 使用 SSE 指令集和 AVX 指令集时明密文分组的切片表示如图 3 和图 4 所示. 以图 3 为例说明实现方法, 先将 8 个分组使用 LOAD 指令加载到寄存器 XMM0 – XMM3, 将每个明文分组看作是 4 个 16 bit 的字, 然后使用 (UN) PACK 指令将每个明文分组中 4 个字分别放在对

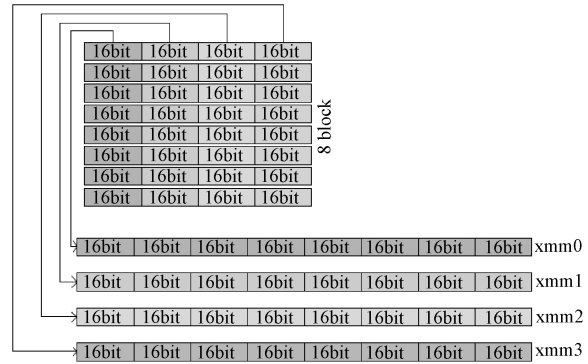


图 3 RECTANGLE 使用 SSE 指令集 8 块并行的切片表示

Fig. 3 Bitslice representation of RECTANGLE in 8-blocks parallel implementation with SSE instruction

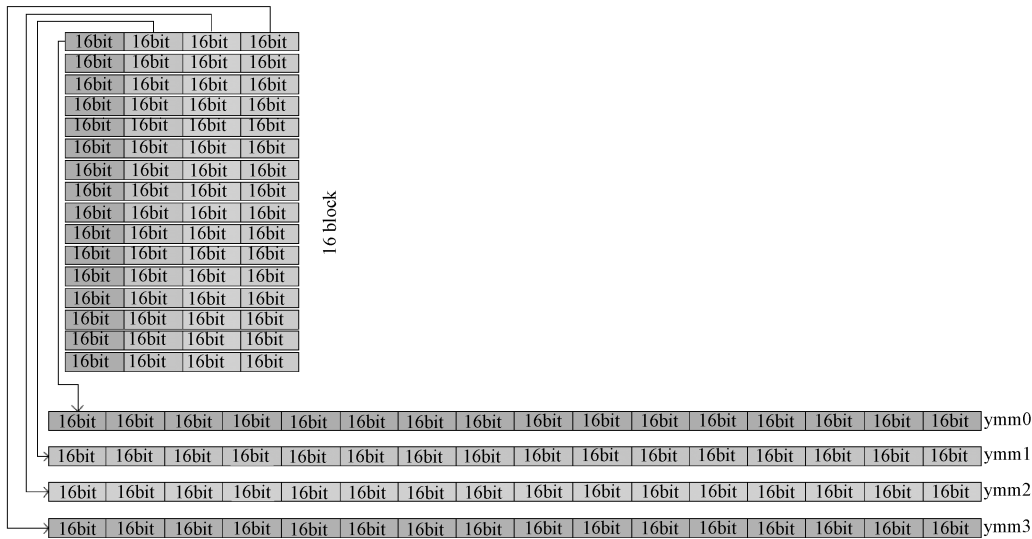


图 4 使用 AVX 指令集 16 块并行的切片表示

Fig. 4 Bitslice representation of RECTANGLE in 16-blocks parallel implementation with AVX instruction

应的寄存器中. 轮密钥加部分, 密钥编排算法将轮密钥存储为 8 个 8 bit 变量, 将其加载到 XMM 寄存器中异或到密码状态. S 盒采用文献[18]中提供的指令序列实现, 行移位部分使用 6 个 SHIFT 和 3 个 XOR 指令实现. 假设 S 盒输入  $A_0A_1A_2A_3$ , 输出为  $B_0B_1B_2B_3$ , 则 RECTANGLE 的 S 盒可以用下列逻辑指令序列实现:

1

T1 = A0⊕A1;

2

T2 = A0|A3;

3

T3 = A2⊕T2;

4

B2 = A1⊕T3;

5

T5 = A0&T3;

6

T6 = A3⊕B2;

7

B1 = T5⊕T6;

8

T8 = ¬B1;

9

T9 = T3|T8;

10

B3 = T1⊕T9;

11

T11 = T8|B3;

12

B0 = T3⊕T11;

其中⊕表示逻辑运算异或, |表示逻辑运算或, &表示逻辑运算与, ¬表示逻辑运算非.

使用 Intel VTune Amplifier 分析 RECTANGLE -80 使用 AVX 指令集加解密 1~1 024 个分组的性能, 结果如表 2 所示. 表 2 中函数名 decrypt、encrypt 分别是指使用 CTR 模式解密和加密过程, time\_dec16、time\_enc16 是指 16 次调用 decrypt、encrypt 函数测试加解密时间, key\_80\_schedule 是指 80 bit 密钥编排. 从表 2 中使用 CPU 时间、停用指令数可以看出, 实现中加密比解密消耗的 CPU 时间多, 80 bit 密钥编排消耗挺多. 80 bit 的密钥

编排性能不好, 如果需实时 (on-the-fly) 进行密钥编排, 还需优化.

表 2 RECTANGLE 使用 AVX 指令集加解密 1~1 024 个分组实现性能

Table 2 Performance of RECTANGLE with AVX instruction for 1~1 024 Blocks

Function name	CPU time/s	Instructions Retired
encrypt	743. 415	4 373 603 100 000
decrypt	718. 999	4 376 677 100 000
key_80_schedule	0. 28	165 300 000

注: Instruction Retired 是指事件执行结束时执行的指令数目.

### 3. 2 实验结果

表 3 是一些轻量级密码算法软件实现速度测试结果, 测试环境为 windows 7 下使用 Intel c++ compiler XE 14. 0 在 Intel (core) i5 -4 570 s CPU 2. 9 GHZ (4 G 内存) 平台, PRESENT、Piccolo、LED 是采用 Ryad 等的代码<sup>[6]</sup>在此平台上测试的结果, 单块加密使用查表方式实现, 8 块或 16 块加密速度使用切片实现. TWINE、SIMON 和 SPECK 的测试结果直接引用文献[16-17]结果, SIMON 和 SPECK 的分组长度和密钥长度有多种, 文中只关注分组长度为 64 bit、密钥长度为 128 bit 的 SIMON 和 SPECK. SEPCK 和 SIMON<sup>[17]</sup>的软件实现速度不包含切片表示与明文、密文之间格式转换的时间, RECTANGLE、LED、Piccolo、PRESENT 的测试速度

表 3 部分轻量级密码算法实验结果

Table 3 Performance of some lightweight block ciphers

Cipher	Key size/bits	Key/cycles	ENC/(cycles/byte)	DEC/(cycles/byte)
RECTANGLE-GR	80/128	857/748	34. 2	30. 9
RECTANGLE-SSE	80/128	865/753	5. 2	5. 1
RECTANGLE-AVX	80/128	866/865	2. 6	2. 5
PRESENT - 80 - 8/16	80	-	21. 7/17. 2	-
PRESENT - 128 - 8/16	128	-	24. 1/18. 3	-
PRESENT - 80/128 - 1	80/128	-	59. 6	-
Piccolo - 80 - 8	80	-	11. 1	-
Piccolo - 128 - 8	128	-	11. 5	-
Piccolo - 80/128 - 1	80/128	-	86. 4/107. 5	-
LED - 64 - 16/32	64	0	19. 1/15. 7	-
LED - 128 - 16/32	128	0	28. 1/23. 1	-
LED - 64 - 1	64	0	63. 5	-
LED - 128 - 1	128	0	94. 8	-
TWINE - 80/128 - 2 <sup>[16]</sup>	80/128	-	5. 55	5. 55
TWINE - 80/128 - 1 <sup>[16]</sup>	80/128	-	11. 10	11. 11
SIMON - 128 - 128 <sup>[17]</sup>	128	-	5. 2	-
SIMON - 128 - 1 <sup>[17]</sup>	128	-	28. 7	-
SPECK - 128 - 8 <sup>[17]</sup>	128	-	2. 4	-
SPECK - 128 - 1 <sup>[17]</sup>	128	-	10. 1	-

注: RECTANGLE-GR、RECTANGLE-SSE、RECTANGLE-AVX 分别是指 RECTANGLE 使用通用指令集、SSE 指令集和 AVX 指令集实现, 最大并行分组数分别为 1、8 和 16. 采用算法-密钥长度-并行分组数的形式为 PRESENT、Piccolo、LED、TWINE、SIMON 和 SPECK 命名. ENC 和 DEC 分别代表加密盒解密速度. Key 代表密钥编排过程中消耗的 CPU 周期数.

包含这个时间.

图 5 和图 6 分别是 RECTANGLE 在上述测试环境的测试结果,使用 SSE 指令集和使用 AVX 指令集方式,在 CTR 模式下不同分组数,RECTANGLE 切片实现中加解密速度的试验结果. X86 平台的结果是在 X64 平台运行 32 位模式测试的.

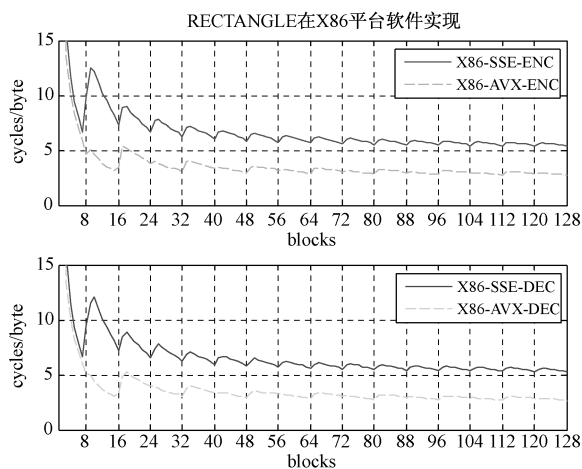


图 5 RECTANGLE 在 Intel (core) i5-4 570 s  
CPU 2.9 GHz (X86) 平台性能

Fig. 5 Performance of RECTANGLE on X86  
platform of Intel (core) i5-4 570 s CPU 2.9 GHz

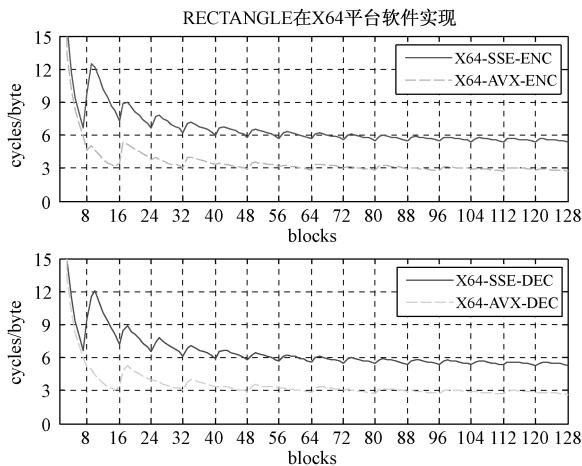


图 6 RECTANGLE 在 Intel (core) i5-4 570 s  
CPU 2.9 GHz (X64) 平台性能

Fig. 6 Performance of RECTANGLE on X64 platform  
of Intel (core) i5-4 570 s CPU 2.9 GHz

在 RECTANGLE 软件实现性能测试时,时间评估函数参考 Gladman 对 AES 测试方法.以密钥编排测试为例说明测试方法,首先测试基准时间,即 2 条 RDTSC 指令中间没有其他指令时的时间.其次在测试密钥编排时,在时间测试之前先执行

一次,消除第一次执行数据载入内存等影响.最后在密钥编排时间测试时,2 条 RDTSC 指令中间执行 8 次加密函数.加密过程和解密过程的测试跟密钥编排过程是一样的,区别是在测试时执行 16 次加密和解密函数.附录包含文中基准时间测试的部分代码和密钥编排性能测试的部分代码,根据这些代码可以很快恢复时间评估函数.

### 3.3 实验结果分析

表 3 中, X64 平台上 RECTANGLE-GR 单块加密速度约为 34.2 cycles/byte,解密速度约为 30.9 cycles/byte. SIMON 和 SPECK<sup>[17]</sup>单块加密速度分别为 28.7 和 10.1 cycles/byte, TWINE<sup>[16]</sup>单块加密速度为 11.10 cycles/byte,相比之下 RECTANGLE 单块加密速度稍慢一些.

在并行处理模式 CTR 加解密过程中,切片实现方式展示了很好的性能优势.从图 5 可以看出,在分组数目大于 128 时,使用 AVX 指令集, RECTANGLE 的加解密速度分别稳定在 2.6 和 2.5 cycles/byte,使用 SSE 指令集加解密速度分别可达 5.2 和 5.1 cycles/byte.使用 AVX 指令集的实现速度比使用 SSE 指令集方式快约 1 倍的速度,因为使用 AVX 指令集使用的是 256 bit 的 YMM 寄存器,比 SSE 中的 128 bit 的 XMM 寄存器大了 1 倍,并行执行的分组数目多 1 倍,切片实现速度跟使用的寄存器的长度及指令集有很大的关系.图 5 和图 6 中的曲线在开始时呈现锯齿状,跟实现方式相关.使用通用指令集方法每次加解密 1 个分组.使用 SSE 指令集方式,当分组数  $\geq 8$  时,先 8 个分组处理;不足 8 个分组 2~7 个分组时,使用 SSE 指令集,仅有 1 个分组使用通用指令集的方法.使用 AVX 指令集方式,当分组数  $\geq 16$  时,先 16 个分组处理;不足 16 个分组且大于 8 个分组时,先使用 SSE 指令集处理 8 个分组;不足 8 个分组时,2~7 个分组使用 SSE 指令集处理,仅有 1 个分组使用通用指令集的方法.选择这样方式的原因是在处理 2~8 个分组时,使用 SSE 指令集要比使用 AVX 指令集所用的指令数目少,处理 1 个分组使用通用指令集是 3 种方式中最快的.

表 3 中 PRESENT 和 LED 的软件实现结果是使用 Ryad 等<sup>[6]</sup>给出的代码在相同的测试平台上测试的,从表 3 数据可以看出,RECTANGLE 的软件实现比 PRESENT、LED 和 Piccolo 更具有优势. SIMON、SPECK 在 X64 平台上加密速度分别可达

5.2 和 2.4 cycles/byte<sup>[17]</sup>,图6中 RECTANGLE 的 SSE 实现在分组大于 128 个分组之后的加密速度稳定在 5.2 cycles/byte. 需要说明的是不同测试方法、不同测试平台会给出不同的测试结果,因此完全公平的比较不同密码算法的软件实现性能非常困难. 此外,由于 TWINE、SIMON 和 SPECK 源代码没有公开,仅给出了 SIMON 和 SPECK 单块加密代码并简单说明两个算法并行处理的实现方法,然而三者测试平台跟文中的测试平台相似,因此表3中直接引用了设计文献[16–17]中的结果. 尽管如此,仍然能够从表3中得到下面的结论:TWINE、RECTANGLE、SIMON 和 SPECK 这4个轻量级分组密码具有优秀的软件实现性能.

## 4 总结

早期的轻量级密码算法更注重硬件实现代价. 自2011年之后,人们发现轻量级密码算法的软件实现同样重要,LED、KLEIN、RECTANGLE、SIMON、SPECK 和 TWINE 等算法的设计都体现了这一趋势. 本文给出 RECTANGLE 切片实现的3种方式,在使用 AVX 指令集方式中加解密速度分别为 2.6 和 2.5 cycles/byte. 后续将研究 SIMON、SPECK 和 TWINE 的软件实现方法,并且在我们平台上进行实际测试,从而对这些轻量级分组密码的软件实现给出更公平合理的比较结果.

### 参考文献

- [1] Moradi A, Poschmann A, Ling A, et al. Pushing the limits: a very compact and a threshold implementation of AES[C]// Paterson G. EUROCRYPT 2011. Berlin Heidelberg: Springer, 2011: 69-88.
- [2] Miroslav K, Ventzislav N, Peter R. Low-latency encryption-Is Lightweight = Light + Wait? [C]//Prouff E, Schaumont P. CHES 2012. Berlin Heidelberg: Springer, 2012: 426–446.
- [3] Nigel Smart (BRIS). ECRYPT II Yearly Report on Standardization (2012–2013) [R/OL]. European: ECRYPT, (2013-01-22) [2014–12]. [http://ec.europa.eu/information\\_society/apps/projects/logos/6/216676/080/deliverables/001\\_DSPA19.pdf](http://ec.europa.eu/information_society/apps/projects/logos/6/216676/080/deliverables/001_DSPA19.pdf).
- [4] Axel Y. Lightweight cryptography-cryptographic engineering for a pervasive world [R/OL]. IACR Cryptology ePrint Archive. (2009) [2014–12]. <http://eprint.iacr.org/2009/516.pdf>.
- [5] Seiichi M, Shiho M. Lightweight cryptography for the cloud: exploit the power of bitslice implementation[C]//Prouff E, Patrick S. CHES 2012. Berlin Heidelberg: Springer, 2012: 408-425.
- [6] Ryad B, Guo J, Victor L, et al. Implementing lightweight block ciphers on x86 architectures [C]//Lange T, Lauter K. SAC 2013. Berlin Heidelberg: Springer, 2014: 324-351.
- [7] Bogdanov A, Knudsen L, Leander G, et al. PRESENT: an ultra-lightweight block cipher [C]//Paillier P, Verbaudhede I. CHES 2007. Berlin Heidelberg: Springer, 2007: 450-466.
- [8] Shamir A. SQUASH: a new MAC with provable security properties for highly constrained devices such as RFID tags [C]//Nyberg K. FSE 2008. Berlin Heidelberg: Springer, 2008: 144-157.
- [9] De C, Dunkelman O, Knežević M. KATAN and KTANTAN: a family of small and efficient hardware-oriented block ciphers [C]//clavier C, Gaj K. CHES 2009. Berlin Heidelberg: Springer, 2009: 272-288.
- [10] Aumasson P, Henzen L, Meier W, et al. Quark: a lightweight hash [C]//Mangard S, Standaert X. CHES 2010. Berlin Heidelberg: Springer, 2010: 1-15.
- [11] Guo J, Peyrin T, Poschmann A. The PHOTON family of lightweight hash functions [C]//Rogaway P. CRYPTO 2011. Berlin Heidelberg: Springer, 2011: 222-239.
- [12] Shibutani K, Isobe T, Hiwatari H, et al. Piccolo: an ultra-lightweight block cipher [C]//Preneel B, Takagi T. CHES 2011. Berlin Heidelberg: Springer, 2011: 342-357.
- [13] Martin A, Benedikt D, Elif K, et al. Block ciphers: focus on the linear layer (feat. PRIDE) [C]//Garay J, Gennaro R. CRYPTO 2014. Berlin Heidelberg: Springer, 2014: 57-76.
- [14] Gong Z, Nikova S, Law Y. KLEIN: a new family of lightweight block ciphers [C]//Juels A, Paar C. RFIDSec 2011. Berlin Heidelberg: Springer, 2012: 1-18.
- [15] Wu W L, Zhang L. LBlock: a lightweight block cipher [C]//Lopez J, Tsudik G. ACNS 2011. Berlin Heidelberg: Springer, 2012: 327-344.
- [16] Suzaki T, Minematsu K, Morioka S, et al. TWINE: a lightweight block cipher for multiple platforms [C]//Knudsen R, Wu H. SAC 2012. Berlin Heidelberg: Springer, 2012: 340-355.
- [17] Ray B, Douglas S, Jason S, et al. The SIMON and SPECK families of lightweight block ciphers [R/OL]. IACR Cryptology ePrint Archive. (2013) [2014-11]. <https://eprint.iacr.org/2013/404.pdf>.
- [18] Zhang W T, Bao Z Z, Lin D D, et al. RECTANGLE: a bit-slice ultra-lightweight block cipher suitable for multiple platforms [R/OL]. IACR Cryptology ePrint Archive. (2014) [2014-11]. <http://eprint.iacr.org/2014/084.pdf>.
- [19] Vincent G, Gaetan L, Fran S, et al. LS-designs: bitslice encryption for efficient masked software implementations [C]//FSE2014. Berlin Heidelberg: Springer, 2014.
- [20] Eisenbarth T, Kumar S, Paar C, et al. A survey of lightweight-cryptography implementations [J]. IEEE Design & Test of Computers, 2007, 24(6): 522-533.



- [21] Thomas, Gong Z, Tim, et al. Compact implementation and performance evaluation of block ciphers in aTiny devices[C] //Mitrokotsa A, Vaudenay S. AFRICACRYPT 2012. Berlin Heidelberg: Springer. 2012; 172-187.
- [22] Matsui M, Murakami Y. Minimalism of software implementation [C] // Moriai S. FSE 2013. Berlin Heidelberg: Springer, 2014; 393-409.
- [23] Ted K, Phillip R. The software performance of authenticated encryption modes [C] // Joux A. FSE 2011. Berlin Heidelberg: Springer, 2011; 306-327.
- [24] Gueron S. Intel advanced encryption standard (AES) instructions set[R/OL]. Intel White Paper Rev3.01. (2012) [2014-11]. <https://software.intel.com/sites/default/files/article/165683/aes-wp-2012-09-22-v01.pdf>.
- [25] Biham E. A fast new DES implementation in software[C] // Biham E. FSE1997. Berlin Heidelberg: Springer, 1997; 260-272.
- [26] Matsui M, Nakajima J. On the power of bitslice implementation on intel Core2 processor [C] // Paillier P, Verbauwhede I. CHES 2007. Berlin Heidelberg: Springer, 2007; 121-134.
- [27] Könighofer R. A fast and cache-timing resistant implementation of the AES[C] // Malkin T. CT-RSA 2008. Berlin Heidelberg: Springer, 2008;187-202.
- [28] Käsper E, Schwabe P. Faster and timing; attack resistant AES-GCM[C] // Clavier C, Gaj K. CHES 2009. Berlin Heidelberg: Springer, 2009;1-17.
- [29] Ross A, Eli B, Lars K. Serpent; a proposal for the advanced encryption standard[R/OL]. NIST AES Proposal. (1998) [2014-11]. <http://cryptosoft.net/docs/Serpent.pdf>.
- [30] Joan D, Michaël P, Gilles A, et al. Nessie proposal; NOEKEON[C/OL]. First Open NESSIE Workshop. (2000) [2014-11]. <http://gro.noekeon.org/Noekeon-spec.pdf>.
- [31] Guido B, Joan D, Michael P, et al. The keccak reference [R/OL]. Submission to NIST(Round 3), 2011[2014-11]. <http://keccak.noekeon.org/>.
- [32] Wu H. SHA-3 proposal JH[R/OL]. Submission to NIST. (2008)[2014-11]. [http://www3.ntu.edu.sg/home/wuhj/research/jh/jh\\_round3.pdf](http://www3.ntu.edu.sg/home/wuhj/research/jh/jh_round3.pdf).

## 附录

### 1) 基准时间测试函数的部分代码:

```
tol = 10; lcnt = sam_cnt = 0;
while( ! sam_cnt) {
    av1 = sig1 = 0.0;
```

```
    for(i = 0; i < SAMPLE1; ++i) {
        cy = (volatile double)read_tsc();
        cy = (volatile double)read_tsc() - cy;
        av1 += cy; sig1 += cy * cy; }
    av1 /= SAMPLE1;
    sig1 = sqrt((sig1 - av1 * av1 * SAMPLE1) /
SAMPLE1);
    sig1 = (sig1 < 0.05 * av1 ? 0.05 * av1 : sig1);
    *av = *sig = 0.0;
    for(i = 0; i < SAMPLE2; ++i) {
        cy = (volatile double)read_tsc();
        cy = (volatile double)read_tsc() - cy;
        if(cy > av1 - sig1 && cy < av1 + sig1) {
            *av += cy; *sig += cy * cy; sam_cnt++; } }
    if(10 * sam_cnt > 9 * SAMPLE2) {
        *av /= sam_cnt; *sig = sqrt(( *sig - *av
* *av * sam_cnt) / sam_cnt);
    if( *sig > (tol/100.0) * *av) { sam_cnt = 0; }
    else { if(lcnt++ == 10) {
        lcnt = 0; tol += 5;
        if(tol > 30) { return FALSE; }
        sam_cnt = 0; } }
```

### 2) 密钥编排测试函数的部分代码:

```
key(userKey[0], ec, k_len);
av1 = sig1 = 0.0;
for(i = 0; i < SAMPLE1; ++i) {
    cy = (double)read_tsc();
    key(userKey[0], ec, k_len);
    key(userKey[1], ec, k_len);
    key(userKey[2], ec, k_len);
    key(userKey[3], ec, k_len);
    key(userKey[4], ec, k_len);
    key(userKey[5], ec, k_len);
    key(userKey[6], ec, k_len);
    key(userKey[7], ec, k_len);
    cy = (double)read_tsc() - cy;
    av1 += cy;
    sig1 += cy * cy; }
av1 /= SAMPLE1;
sig1 = sqrt((sig1 - av1 * av1 * SAMPLE1) /
SAMPLE1);
```