软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(10):4827−4845 [doi: 10.13328/j.cnki.jos.007316] [CSTR: 32375.14.jos.007316]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

FBS-uBlock: 灵活的 uBlock 算法比特切片优化方法*

龚子睿[1,2], 郭 华[1,2], 陈 晨[1], 张宇轩[1], 陈俊鑫[1], 关振宇[1]

[1](北京航空航天大学 网络空间安全学院, 北京 100191)

[2](复杂关键软件环境全国重点实验室 (北京航空航天大学), 北京 100191)

通信作者: 郭华, E-mail: hguo@buaa.edu.cn

摘 要: uBlock 算法在算法设计、侧信道防护、物联网应用、密码分析领域得到了广泛应用. 虽然 uBlock 算法适合高速实现, 但目前该算法公开的实现速率远不如 AES、SM4 等算法. 比特切片是优化分组密码的常用方法, 但在采用比特切片优化 uBlock 算法时, 面临着因寄存器资源不足而导致的巨大访存开销问题. 为 uBlock 算法设计了一种灵活的比特切片优化方法 FBS-uBlock (flexible bit slicing uBlock), 降低算法在比特切片下占用的寄存器数量, 进而降低访存开销, 提升速率. 经过测试, 该优化方法最多能够让 uBlock-128/128、uBlock-128/256 和 uBlock-256/256 算法的访存指令分别降低 71%、71% 和 72%, 加密速率最高能够分别达到 12 758 Mb/s、8 944 Mb/s 和 8 984 Mb/s, 比设计文档中的实现速率分别提升了 3.9、4.2 和 3.4 倍.

关键词: 分组密码; uBlock 算法; 软件优化; 比特切片; 单指令多数据

中图法分类号: TP309

中文引用格式: 龚子睿, 郭华, 陈晨, 张宇轩, 陈俊鑫, 关振宇. FBS-uBlock: 灵活的uBlock算法比特切片优化方法. 软件学报, 2025, 36(10): 4827–4845. http://www.jos.org.cn/1000-9825/7316.htm

英文引用格式: Gong ZR, Guo H, Chen C, Zhang YX, Chen JX, Guan ZY. FBS-uBlock: Flexible Bit Slicing Optimization Method of uBlock Algorithm. Ruan Jian Xue Bao/Journal of Software, 2025, 36(10): 4827–4845 (in Chinese). http://www.jos.org.cn/1000-9825/7316.htm

FBS-uBlock: Flexible Bit Slicing Optimization Method of uBlock Algorithm

GONG Zi-Rui[1,2], GUO Hua[1,2], CHEN Chen[1], ZHANG Yu-Xuan[1], CHEN Jun-Xin[1], GUAN Zhen-Yu[1]

[1](School of Cyber Science and Technology, Beihang University, Beijing 100191, China)

[2](State Key Laboratory of Complex & Critical Software Environment (Beihang University), Beijing 100191, China)

Abstract: The uBlock algorithm has been widely used in algorithm design, side channel protection, Internet of Things applications, and cryptanalysis. Although the uBlock algorithm is suitable for high-speed implementation, the publicly available implementation rate of this algorithm is far lower than that of algorithms such as AES and SM4. Bit slicing is a common method to optimize block ciphers. However, when using bit slicing to optimize the uBlock algorithm, it faces the problem of huge memory access overhead due to insufficient register resources. In this study, a flexible bit slicing optimization method named FBS-uBlock is designed for the uBlock algorithm. It reduces the number of registers occupied by the algorithm under bit slicing, thus reducing the memory access overhead and improving the speed. After testing, the proposed optimization method can reduce the memory access instruction of uBlock-128/128, uBlock-128/256, and uBlock-256/256 algorithms by up to 71%, 71%, and 72%, respectively. The maximum encryption rates can reach 12 758 Mb/s, 8 944 Mb/s, and 8 984 Mb/s respectively, which are 3.9, 4.2, and 3.4 times higher than the implementation rates in the design documentation.

Key words: block cipher; uBlock algorithm; software optimization; bit slicing; single instruction multiple data (SIMD)

---

* 基金项目: 北京市自然科学基金 (4242022); CCF-绿盟科技“鲲鹏”科研基金 (CCF-NSFOCUS 2023006); 大学生创新创业训练计划 (X202210006242)
收稿时间: 2024-02-06; 修改时间: 2024-10-08; 采用时间: 2024-10-29; jos 在线出版时间: 2025-07-09
CNKI 网络首发时间: 2025-07-10

uBlock[1]是一族分组密码算法, 自提出以来就受到了广泛关注, 在算法设计[2,3]、侧信道防护[4]、物联网应用[5]和密码分析领域[6−10]都有相关研究工作. uBlock 算法在设计之初就考虑了处理器的计算资源, 采用的 4 比特 S 盒在软件平台可利用 SSE、NEON 指令集高效实现, 但 uBlock 算法目前的实现速率仍不如 AES、SM4 等分组密码算法.

比特切片是分组密码算法常用的优化方法. 自 Biham[11]提出比特切片思想后, 学者们对比特切片进行了广泛研究, 包括将比特切片应用在各种算法的优化中、提出比特切片的变体、让设计的算法天然具备比特切片的形式等. 当将 Biham 的比特切片方法用于分组密码算法实现时, 扩散运算使得采用比特切片优化的算法不满足程序设计的局部性原则, 从而产生大量的访存开销. 例如, 当使用类似 Biham 的比特切片方法优化 uBlock 算法时, uBlock-128/128 和 uBlock-128/256 至少需要占用 128 个寄存器, uBlock-256/256 至少需要占用 256 个寄存器, 但通用处理器仅有 16 个或 32 个寄存器. 由于密码算法中所有的算术运算都是在处理器的寄存器上计算, 倘若算法需要使用的寄存器数量大于处理器提供的物理寄存器数量, 则会产生额外的访存开销. 因为访存指令比一般的算术指令更耗时, 所以应该尽量减少比特切片的访存指令. 虽然目前学术界也有针对其他密码算法的访存优化方法和比特切片优化方法, 但用于优化 uBlock 算法的比特切片实现时但仍存在以下问题.

(1) 在优化 uBlock 算法比特切片实现带来的访存开销时, 发现 AES 或 SM4 算法的切片表示方法并不适合 uBlock 算法轮函数中的运算, 因此无法直接使用 Könighofer[12]、Käsper 等人[13]和陈晨等人[14]的切片表示方法, 需要针对 uBlock 算法重新设计新的切片表示方法.

(2) 目前公开的 uBlock 算法 S 盒逻辑函数使用了硬件平台特有的同或、与非、或非门, 这些逻辑运算在软件平台上需要多条处理器指令组合才能完成, 因此现有的逻辑函数不适合比特切片软件实现, 降低 S 盒在比特切片时的计算开销需要以降低比特切片门复杂度 (bitslice gate complexity, BGC) 为目标优化 S 盒.

针对上述问题 (1), 本文通过分析 uBlock 算法轮函数结构来总结设计新的比特切片表示法的指导思想, 依据指导思想推出多种不同寄存器占用数量的 uBlock 算法比特切片表示法, 减少算法计算过程需要的访存指令, 灵活适用于各种拥有不同寄存器数量的处理器平台. 针对上述问题 (2), 本文利用现有的 Lighter[15]工具优化 uBlock 算法 S 盒, 限制最终的逻辑函数仅能使用软件处理器支持的逻辑运算, 以减少比特切片门复杂度为目标搜索更适合比特切片实现的 S 盒逻辑函数. 本文主要工作如下.

(1) 提出了一种灵活的 uBlock 算法比特切片优化方法 FBS-uBlock (flexible bit slicing uBlock), 降低算法在比特切片下占用的寄存器数量, 进而降低实现时的访存开销. 基于该方法, 对于 uBlock-128/128 和 uBlock-128/256, 本文提出了占用 128、64、32、16 和 8 个寄存器的比特切片表示法; 对于 uBlock-256/256, 本文提出了占用 256、128、64、32、16 和 8 个寄存器的比特切片表示法.

(2) 给出了适合比特切片软件实现的 uBlock 算法 S 盒逻辑函数, 降低了算法 S 盒和逆 S 盒的比特切片门复杂度. 本文采用 Lighter 工具完成 S 盒的逻辑函数化简, 得到 BGC 为 9 的 S 盒逻辑函数以及 BGC 为 9 的逆 S 盒逻辑函数, 优于设计文档中 BGC 为 14 的 S 盒逻辑函数.

(3) 基于上述优化方法在 x86 平台实现了 uBlock 算法, 并进行了理论分析和实验分析. 本文基于所提出的优化方法分别实现了 uBlock-128/128、uBlock-128/256 和 uBlock-256/256 算法, 与设计文档相比, 加解密速率分别提升了 3.9、4.2 和 3.4 倍; 与 Biham 的比特切片表示法相比, 访存指令条数最多能够分别降低 71%、71% 和 72%, 访存指令占比从 60%、61% 和 62% 最多能够降低到 25%、25% 和 24%.

本文第 1 节介绍相关工作. 第 2 节介绍背景知识, 包括 uBlock 算法、比特切片方法和单指令多数据指令集. 第 3 节介绍本文提出的优化方法 FBS-uBlock, 包括算法轮函数结构分析、比特切片表示法、优化非线性 S 盒这 3 个部分. 第 4 节介绍实现的细节和理论分析. 第 5 节给出性能测试和分析. 第 6 节总结全文.

1 相关工作

比特切片是分组密码算法常用的优化方法. 自 Biham[11]提出比特切片思想后, 学者们对比特切片进行了广泛研究, 包括将比特切片应用在各种算法的优化中、提出比特切片的变体、设计比特切片友好的算法. 1997 年, Biham 在优化 DES 算法时使用了比特切片思想, 将 DES 加解密速率提升了 64%. 之后 Kwan[16]、May 等人[17]进

一步优化了 DES 算法, Kwan 利用选择函数优化了 S 盒的计算, May 利用 Schaumüller-Bichl[18]提供的 S 盒逻辑表达式加速了算法. Matsuda 等人[19]、Papapagiannopoulos[20]采用比特切片优化了 PRESENT 算法, 张笑从等人[21]、王磊等人[22]、陈晨等人[23]采用比特切片优化了 SM4 算法, 并降低了计算 S 盒时需要的逻辑门个数. 对于 AES 算法, 还有众多针对 CPU 平台[24−28]和 GPU 平台[29,30]的比特切片优化.

在 Biham 所提出的比特切片方法中, 每个寄存器代表一个比特位, 后续也产生了不一样的比特切片表示法. 例如, Könighofer[12]考虑到 CPU 中仅有 16 个 64 比特寄存器, 已有的 AES 比特切片方法无法将数据全部保存在寄存器中, 导致产生了大量影响性能的访存指令, 由此提出了访存开销更低的 AES 比特切片表示法. Käsper 等人[13]考虑 XMM 寄存器的数量, 采用与 Könighofer 类似的比特切片方法优化了 AES 算法的比特切片实现. 苗鑫等人[31,32]将 SM4 算法的明文数据视为立方体的结构, 提出新的针对 SM4 算法的比特切片表示法. 王闯等人[33]提出一种通用的切片分组密码算法模型, 所优化的 SM4 算法在切片形式上与苗鑫等人的方法类似, 但其重点考虑的是减少数据切片过程的开销. 陈晨等人[14]针对传统比特切片的访存问题, 设计了基于寄存器的优化方法, 并从指令层面分析了优化前后的内存读取次数和耗时. Adomnicai 等人[34]提出针对 AES 算法的固定切片优化方法, 将行移位和列混淆融合, 并利用 ARM 处理器的桶形移位器削减移位运算的开销. 此外, 一些密码算法在设计之初就考虑了比特切片实现友好的性质, 例如 GIFT 算法[35]、ASCON 算法[36]、FESH 算法[37]、TANGRAM 算法[38]等. 对于 PRESENT 算法, 虽然原始结构不是比特切片实现友好的形式, 但 Reis 等人[39]提出了线性层分解方法, 将 PRESENT 算法转变成了易于比特切片实现的形式, 从而可通过比特切片方法进行实现加速.

单指令多数据 (single instruction multiple data, SIMD) 指令集是加速密码算法实现的常用技术, 常见的 SIMD 指令集有 x86 架构的 SSE 指令集、AVX 指令集和 ARM 架构的 NEON 指令集. Hamburg[40]利用 Permute 指令加速 AES 算法实现, CBC 模式加密速率能够达到 10.8 c/B (cycles/byte), CTR 模式能够达到 5.4 c/B. Matsuda 等人[19]使用 AVX 指令集优化 PRESENT 算法和 Piccolo 算法, PRESENT-80/128 达到 5.79 c/B, Piccolo-80 达到 5.69 c/B. Seo 等人[41]使用 NEON 指令集优化 LEA 算法, 4 线程并行能够达到 3.2 c/B. Park 等人[42]使用 NEON 指令集优化 SIMON/SPECK 算法, SIMON-128/128 能够达到 32.4 c/B, SPECK-128/128 能够达到 9.7 c/B, 多线程下分别达到 14.3 c/B 和 5.1 c/B. Adomnicai 等人[43]采用 S 盒分解方法优化 Skinny-128 算法, 在 ARM 平台达到 127 c/B, 在 x86 平台达到 44 c/B. 对于 Banik 等人[35]提出的 GIFT 算法, GIFT-64/128 在 AVX2 指令集的优化下能够达到 2.10 c/B. 对于 uBlock 算法[1], 在使用 x86 平台的 SSE 指令集优化下, uBlock-128/128 算法能够达到 11.8 c/B, uBlock-128/256 算法能够达到 17.4 c/B, uBlock-256/256 算法能够达到 13.6 c/B. 高莹等人[44]使用 AVX2 指令集优化 uBlock 算法, 比设计文档中的实现方法提升了 269%、182% 和 49%.

密码算法中所有的算术运算都是在处理器的寄存器上计算, 倘若算法需要使用的寄存器数量大于处理器提供的物理寄存器数量, 则会产生访存开销. 研究者们进行了各种尝试减少访存开销. 在公钥密码算法中, Hutter 等人[45,46]和 Seo 等人[47]提出新的高精度乘法计算方法, 减少读取指令的条数. 在分组密码算法中, Schwabe 等人[27]通过设计寄存器调度算法优化 AES 算法, 减少 S 盒在比特切片下消耗的访存指令; Könighofer[12]和 Käsper 等人[13]针对算法结构提出了新的比特切片表示法, 减少寄存器使用数量; May 等人[17]在优化 DES 算法时发现, Schaumüller-Bichl[18]的 S 盒表达式逻辑门个数比 Kwan[16]的更多, 但使用前者优化的 DES 算法实现速率反而比后者的更快, 他们认为是访存导致了这一反常现象.

2 基础知识

2.1 uBlock 算法

uBlock 是一族分组密码算法, 整体结构是 PX 结构, 根据分组和密钥的比特长度可分为 uBlock-128/128、uBlock-128/256 和 uBlock-256/256, 迭代轮数分别为 16、24 和 24. 图 1 展示了 uBlock 算法的加密轮函数, 加密输入 n 比特明文 X, 每轮迭代中依次计算轮密钥加、S 盒、线性变换 B 和线性变换 PL/PR, 输出前再计算一次轮密钥加. 解密是加密的逆过程, 在此不赘述.

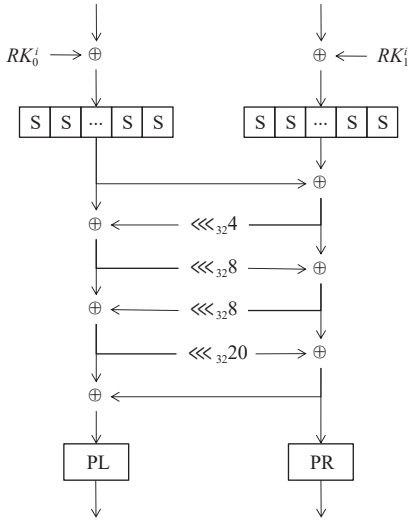

图 1　uBlock 算法加密轮函数

uBlock 算法采用 4 比特的 S 盒, S 盒以及 S 盒的逆如表 1 所示.

表 1　uBlock 算法 4 比特 S 盒与 S 盒的逆

x	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
$s(x)$	7	4	9	c	b	a	d	8	f	e	1	6	0	3	2	5
$s^{-1}(x)$	c	a	e	d	1	f	b	0	7	2	5	4	3	6	9	8

uBlock 算法的 S 盒硬件逻辑门结构如图 2 所示, 在硬件实现中共需 8 个逻辑门, 即 2 个与非门、2 个或非门、2 个异或门、2 个异或非门. 算法的 S 盒的关键路径为 4, 每比特平均 1 个乘法电路.

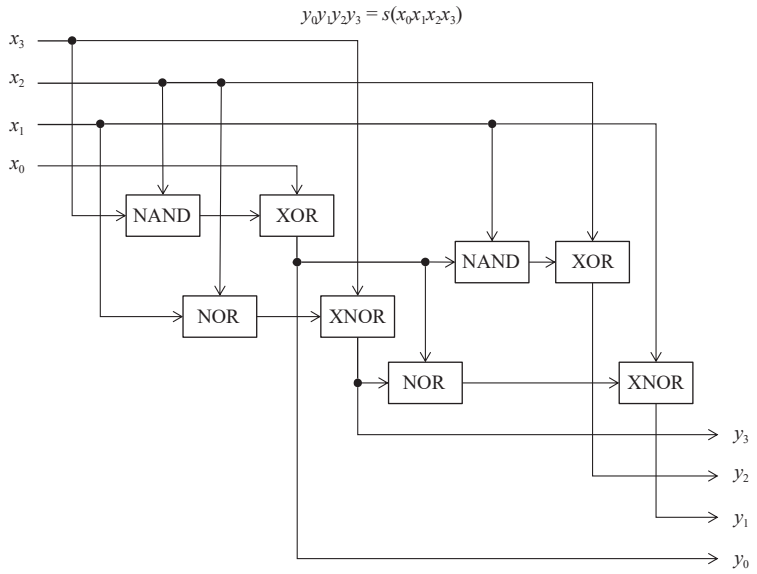

图 2　uBlock 算法 S 盒硬件逻辑门

uBlock 算法的 PL 和 PR 是 $n/16$ 个字节的向量置换, 不同分组比特长度所对应的置换表和逆置换表如表 2 所示.

表 2　uBlock 算法 PL/PR 置换表和逆置换表

x	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
PL_{128}	1	3	4	6	0	2	7	5	—	—	—	—	—	—	—	—
PR_{128}	2	7	5	0	1	6	4	3	—	—	—	—	—	—	—	—
PL_{128}^{-1}	4	0	5	1	2	7	3	6	—	—	—	—	—	—	—	—
PR_{128}^{-1}	3	4	0	7	6	2	5	1	—	—	—	—	—	—	—	—
PL_{256}	2	7	8	13	3	6	9	12	1	4	15	10	14	11	5	0
PR_{256}	6	11	1	12	9	4	2	15	7	0	13	10	14	3	8	5
PL_{256}^{-1}	15	8	0	4	9	14	5	1	2	6	11	13	7	3	12	10
PR_{256}^{-1}	9	2	6	13	5	15	0	8	14	4	11	1	3	10	12	7

2.2　比特切片方法

在 Biham[11]的比特切片方法中, 数据分组的每个比特位会被分散存储在不同寄存器中, 每个寄存器代表一个比特. 对于分组密码算法, 比特切片很适合处理算法中比特粒度的运算, 尤其是线性部分中以比特为单位的置换和异或运算.

图 3 是比特切片的示意图, 如果处理器提供 n 比特长的寄存器, 那么采用比特切片优化后的分组密码算法需要一次性读入 n 组数据, 然后将每组数据的第 i 号比特存放到第 i 号寄存器中. 以分组长度是 128 比特的密码算法为例, 这种切片方法总共需要 128 个寄存器.

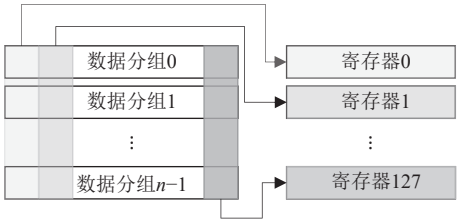

图 3　比特切片示例图

数据分组的比特位开始连续存储在内存中, 比特切片需要将它们分散存储在不同寄存器中. 本文将这种对数据处理的过程称为数据编排. 将 n 个数据分组按行排列成 n 行, 分组的 k 个比特位视作 k 列, 数据编排可以看成 $n\times k$ 的矩阵转置, 转置后变成 k 行 (k 个寄存器) 和 n 列 (每个寄存器 n 比特长). 算法 1 是比特切片的数据编排算法, 以 n 阶方阵为例, 算法复杂度是 $n/2\times\log_2(n)$.

算法 1. 比特切片数据编排算法.

输入: $n\times n$ 的数据矩阵 $\boldsymbol{X}$;
输出: 转置后的 $n\times n$ 的数据矩阵 $\boldsymbol{X}$.

1. **for** $j = 0$ **to** $\log_2(n) - 1$ **do**
2. 　　$k \leftarrow 2^j$
3. 　　$m_j \leftarrow 2^k$ 个比特 0 拼接上 2^k 个比特 1
4. 　　$m_j' \leftarrow \neg m_j$
5. 　　**for** i=0 **to** $n/2 - 1$ **do**
6. 　　$r \leftarrow 2\times(i - (i \bmod k)) + (i \bmod k)$
7. 　　$t \leftarrow (\boldsymbol{X}[r] \wedge m_j') \vee ((\boldsymbol{X}[r+k] \wedge m_j') >> k)$

8. $\boldsymbol{X}[r+k] \leftarrow ((\boldsymbol{X}[r] \wedge m_j) << k) \vee (\boldsymbol{X}[r+k] \wedge m_j)$
9. $\boldsymbol{X}[r] = t$
10. **end**
11. **end**

2.3 SIMD 指令集

SIMD 指令集是一种提升算法性能的重要技术, 其并行处理能力能够让处理器在同一时间处理多份数据, 被广泛应用于对称密码和非对称密码的软件实现中. 表 3 介绍了 x86 架构的 AVX2 指令集和 ARM 架构的 NEON 指令集, 其中 AVX2 指令集操作 256 比特长的寄存器, NEON 指令集操作 128 比特长的寄存器.

表 3 x86 和 ARM 指令集中的指令以及对应关系

指令简介	AVX2指令	NEON指令
数据加载	_mm256_loadu_si256	vld1q_u8
数据存储	_mm256_storeu_si256	vst1q_u8
逻辑与	_mm256_and_si256	vandq_u8
逻辑或	_mm256_or_si256	vorrq_u8
逻辑异或	_mm256_xor_si256	veorq_u8
64比特逻辑左移	_mm256_slli_epi64	vqshlq_n_u64
64比特逻辑右移	_mm256_srli_epi64	vrshrq_n_u64
数据重排/4比特查表	_mm256_shuffle_epi8	vqtbl1q_u8
64比特高位打包	_mm256_unpackhi_epi64	vzip2q_u64
64比特低位打包	_mm256_unpacklo_epi64	vzip1q_u64
128比特重排	_mm256_permute2x128_si256	—

3 灵活的 uBlock 算法比特切片优化方法 FBS-uBlock

3.1 算法轮函数结构分析

本文减少比特切片访存开销的核心思想是通过让每个寄存器表示多个 uBlock 算法的比特位来减少寄存器的使用. 使用 Biham 的比特切片方法优化 uBlock 算法时, 每个寄存器表示 1 个比特位, 占用的寄存器数量和 uBlock 算法的分组比特数相同. 本文尝试让每个寄存器表示 uBlock 算法的 2 个、4 个、8 个和 16 个比特位, 从而使得占用寄存器数量降低至之前的 1/2、1/4、1/8 和 1/16. 但一个寄存器表示多个比特位的方式会让数据中比特位与比特位之间的联系变得更紧密, 因此会增加线性变换中不同比特位之间的运算开销. 本文针对 uBlock 算法的结构, 充分考虑线性变换的结构特点, 精心设计了多种比特切片表示法, 在削减寄存器数量的同时尽量减少额外的运算开销. 下面从算法迭代过程的数据结构、轮密钥加、S 盒、线性变换 B 和线性变换 PL/PR 这些模块详细分析, 讨论减少寄存器的可行性, 并说明本文是如何减少额外运算开销的.

uBlock 算法加密和解密的轮函数迭代分成了 X_0 和 X_1 左右两部分, 线性变换 B 是 X_0 和 X_1 相互的循环移位与异或, 迭代最后一步的 PL 和 PR 置换分别对 X_0 和 X_1 进行. 因此, 本文将 X_0 和 X_1 这两部分占用的寄存器分别考虑, 在减少寄存器占用时不再针对算法分组的整体, 而是针对 X_0 和 X_1 这两部分.

uBlock 算法的轮密钥加模块是异或运算, 是对应比特位的模 2 加法, 不存在不同比特位之间的运算. 因此, 任何比特切片表示法都不会对该模块的计算造成影响, 只需要预先把轮密钥的表示形式转变成和明文一致的表示形式即可.

uBlock 算法的非线性变换由多个相同的 4 比特的 S 盒并置而成, 这意味着一个分组内的多个 S 盒可以同时计算. 另外, 比特切片中的 S 盒需要通过逻辑门函数计算, 包含大量的不同比特位之间的复杂运算, 所以需要让输入 S 盒的这 4 个比特位分散在不同寄存器之中, 否则会极大增加 S 盒计算过程的额外开销.

uBlock 算法的线性变换 B 中主要的运算是 32 比特循环移位, X_0 和 X_1 在进行循环移位时会被分成数段 32 比特数据, 每段独立执行循环移位. 因此, 在线性变换 B 中可以从两个角度减少占用的寄存器数量, 一个角度是针对 32 比特循环移位模块的内部, 在表示 32 比特的数据时减少寄存器的占用; 另一个角度是针对并行的多个 32 比特循环移位模块之间, 通过按行并置多个模块来减少寄存器占用. 因为"针对模块内"增加了循环移位指令的代价, 而"针对模块间"没有额外代价, 故在减少寄存器占用数量时首先采用"针对模块间"的方式.

针对循环移位模块内, 如果将 32 比特数据写成列向量的形式, 异或运算变成模 2 加法, 循环移位转变成矩阵乘法 $\boldsymbol{M}_{\mathrm{ROL}}\times x$ 的形式, 那么 $x'=x\oplus(y\lll 4)$ 可以表示成公式 (1) 的形式. 通过观察公式 (1) 中矩阵 $\boldsymbol{M}_{\mathrm{ROL}}$ 的值可以发现, 矩阵 $\boldsymbol{M}_{\mathrm{ROL}}$ 恰好是每一行循环右移 4 位的单位矩阵.

$$x'=x\oplus\boldsymbol{M}_{\mathrm{ROL}}\times y\Leftrightarrow\begin{bmatrix}x'_0\\x'_1\\x'_2\\\vdots\\x'_{29}\\x'_{30}\\x'_{31}\end{bmatrix}=\begin{bmatrix}x_0\\x_1\\x_2\\\vdots\\x_{29}\\x_{30}\\x_{31}\end{bmatrix}+\begin{bmatrix}00010000\ldots\\00001000\ldots\\00000100\ldots\\\vdots\\01000000\ldots\\00100000\ldots\\00010000\ldots\end{bmatrix}\times\begin{bmatrix}y_0\\y_1\\y_2\\\vdots\\y_{29}\\y_{30}\\y_{31}\end{bmatrix}\tag{1}$$

进一步观察可以发现, 矩阵 $\boldsymbol{M}_{\mathrm{ROL}}$ 是一个循环矩阵, 如果把 $\boldsymbol{M}_{\mathrm{ROL}}$ 写成 2×2、4×4、8×8 或 16×16 的分块矩阵后, 分块矩阵也是循环矩阵. 因此, 如果将数据写成一个二维的矩阵, 将循环移位运算对应的矩阵 $\boldsymbol{M}_{\mathrm{ROL}}$ 写成分块矩阵的形式, 那么公式 (1) 将转变成公式 (2)–(4) 的形式. 为了简化表达式, 在公式 (2)–(4) 中使用符号 x_{i-j} 代表由 $x_i, x_{i+1}, \ldots, x_j$ 组成的列向量.

$$\begin{aligned}\begin{bmatrix}x'_{0-15}\\x'_{16-31}\end{bmatrix}&=\begin{bmatrix}x_{0-15}\\x_{16-31}\end{bmatrix}+\begin{bmatrix}\boldsymbol{A}&\boldsymbol{B}\\\boldsymbol{B}&\boldsymbol{A}\end{bmatrix}\times\begin{bmatrix}y_{0-15}\\y_{16-31}\end{bmatrix}\Leftrightarrow\begin{bmatrix}x'_{0-15}&x'_{16-31}\end{bmatrix}\\&=\begin{bmatrix}x_{0-15}&x_{16-31}\end{bmatrix}+[\boldsymbol{A}]\times\begin{bmatrix}y_{0-15}&y_{16-31}\end{bmatrix}+[\boldsymbol{B}]\times\begin{bmatrix}y_{16-31}&y_{0-15}\end{bmatrix}\end{aligned}\tag{2}$$

$$\begin{aligned}\begin{bmatrix}x'_{0-7}\\x'_{8-15}\\x'_{16-23}\\x'_{24-31}\end{bmatrix}&=\begin{bmatrix}x_{0-7}\\x_{8-15}\\x_{16-23}\\x_{24-31}\end{bmatrix}+\begin{bmatrix}\boldsymbol{A}&\boldsymbol{B}&\boldsymbol{C}&\boldsymbol{D}\\\boldsymbol{D}&\boldsymbol{A}&\boldsymbol{B}&\boldsymbol{C}\\\boldsymbol{C}&\boldsymbol{D}&\boldsymbol{A}&\boldsymbol{B}\\\boldsymbol{B}&\boldsymbol{C}&\boldsymbol{D}&\boldsymbol{A}\end{bmatrix}\times\begin{bmatrix}y_{0-7}\\y_{8-15}\\y_{16-23}\\y_{24-31}\end{bmatrix}\Leftrightarrow\begin{bmatrix}x'_{0-7}&x'_{8-15}&x'_{16-23}&x'_{24-31}\end{bmatrix}\\&=\begin{bmatrix}x_{0-7}&x_{8-15}&x_{16-23}&x_{24-31}\end{bmatrix}+[\boldsymbol{A}]\times\begin{bmatrix}y_{0-7}&y_{8-15}&y_{16-23}&y_{24-31}\end{bmatrix}\\&\quad+[\boldsymbol{B}]\times\begin{bmatrix}y_{8-15}&y_{16-23}&y_{24-31}&y_{0-7}\end{bmatrix}+[\boldsymbol{C}]\times\begin{bmatrix}y_{16-23}&y_{24-31}&y_{0-7}&y_{8-15}\end{bmatrix}\\&\quad+[\boldsymbol{D}]\times\begin{bmatrix}y_{24-31}&y_{0-7}&y_{8-15}&y_{16-23}\end{bmatrix}\end{aligned}\tag{3}$$

$$\begin{aligned}\begin{bmatrix}x'_{0-3}\\x'_{4-7}\\\vdots\\x'_{28-31}\end{bmatrix}&=\begin{bmatrix}x_{0-3}\\x_{4-7}\\\vdots\\x_{28-31}\end{bmatrix}+\begin{bmatrix}\boldsymbol{A}&\boldsymbol{B}&\ldots&\boldsymbol{H}\\\boldsymbol{H}&\boldsymbol{A}&\ldots&\boldsymbol{G}\\\vdots&\vdots&\ddots&\vdots\\\boldsymbol{B}&\boldsymbol{C}&\ldots&\boldsymbol{A}\end{bmatrix}\times\begin{bmatrix}y_{0-3}\\y_{4-7}\\\vdots\\y_{28-31}\end{bmatrix}\Leftrightarrow\begin{bmatrix}x'_{0-3}&x'_{4-7}&\ldots&x'_{28-31}\end{bmatrix}\\&=\begin{bmatrix}x_{0-3}&x_{4-7}&\ldots&x_{28-31}\end{bmatrix}+[\boldsymbol{A}]\times\begin{bmatrix}y_{0-3}&y_{4-7}&\ldots&y_{28-31}\end{bmatrix}\\&\quad+[\boldsymbol{B}]\times\begin{bmatrix}y_{4-7}&\ldots&y_{28-31}&y_{0-3}\end{bmatrix}+\ldots+[\boldsymbol{H}]\times\begin{bmatrix}y_{28-31}&y_{0-3}&\ldots&y_{24-27}\end{bmatrix}\end{aligned}\tag{4}$$

观察公式 (2)–(4) 等号右侧数据 y 的表示, 可以发现公式对数据进行了循环的变换, 按照列的单位进行了循环左移. 可以验证, 对于 32 比特数的循环 4、8、20 移位运算, 都可以有类似的表示方式. 上述理论分析表明, 可以通过引入一些循环移位运算来降低列向量的行数. 在比特切片的表示中, 数据每一行代表一个寄存器, 行数的降低意味着寄存器占用数量的减少, 从而能够达到减少访存开销的目的.

对于线性变换 B 的运算 $x'=x\oplus(y\lll s)$, $s\in\{0, 4, 8, 20\}$, 将数据 x 和 y 都表示成 $k\times(32/k)$ 的矩阵 $\boldsymbol{M}_x$ 和 $\boldsymbol{M}_y$, 用符号 $\mathrm{Reg}_i^{(x)}$ 表示 x 的比特切片表示下的第 i 个寄存器, 所有表示形式的计算方式可通过公式 (5) 概括. 数据 x 第 i

比特在矩阵 $\boldsymbol{M}_x$ 中的行列坐标是 (i mod k, i/k), y<<<s 的第 i 比特在矩阵 $\boldsymbol{M}_y$ 中的行列坐标是 ((i+s) mod k, (i+s)/k). 为了执行 (i mod k, i/k) 和 ((i+s) mod k, (i+s)/k) 这两个比特位的异或, 行坐标的差异通过改变寄存器的索引抵消, 列坐标的差异 $\Delta=(i+s)/k-i/k$ 通过循环左移抵消.

$$\mathrm{Reg}_i^{(x')}=\mathrm{Reg}_i^{(x)}\oplus\left(\mathrm{Reg}_{(i+s)\bmod k}^{(y)}<<<\left(\left\lfloor\frac{i+s}{k}\right\rfloor-\left\lfloor\frac{i}{k}\right\rfloor\right)\right),\ i\in[0,k-1] \tag{5}$$

图 4 是其中一种循环移位情况的示例. 假定矩阵 k=8, 循环移位的位数 s=20, 那么移位后序号 i 为 2 的寄存器 $\mathrm{Reg}_2^{(y<<<20)}$ 可通过令移位前序号为 6 的寄存器 $\mathrm{Reg}_6^{(y)}$ 循环左移 2 个单位来计算.

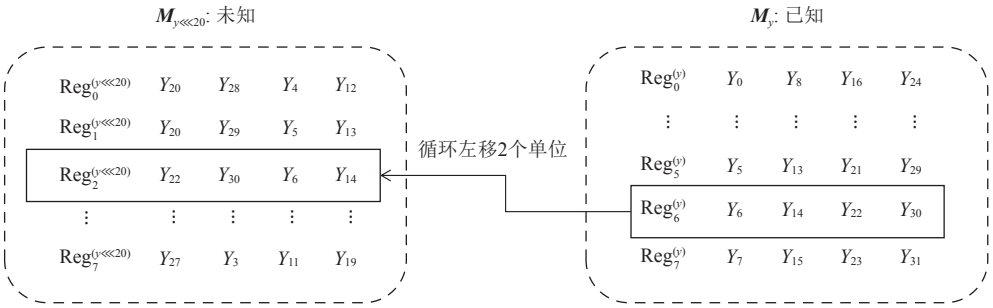

图 4 循环移位计算示例

针对循环移位模块间, 当分组比特是 128 比特时, X_0 (和 X_1) 是 64 比特长度, 这意味着 X_0 (和 X_1) 可以从 64×1 的列向量表示成 32×2 的二维矩阵形式; 当分组比特是 256 比特时, X_0 (和 X_1) 是 128 比特长度, 表示 X_0 (和 X_1) 可以从 128×1 的列向量表示成 64×2 或 32×4 的二维矩阵形式. 这从另一角度减少了比特切片所占用的寄存器数量.

uBlock 算法的 PL 和 PR 两个模块都是字节粒度的重排操作, 字节重排会让不同字节比特位的相对位置发生改变, 所以切片后的一个字节内不应该包含 2 个不同字节的比特位, 让切片后的数据仍然能够以字节为单位进行重排, 进而使用 SIMD 指令集中的重排指令快速实现.

3.2 比特切片表示法 FBS-uBlock

经过第 3.1 节的分析, 可以得到下面 4 个要点, 用于指导 uBlock 算法比特切片表示法的设计.

(1) 通过考虑迭代过程的数据结构, 说明在减少寄存器占用数量时应分别对 X_0 和 X_1 进行.

(2) 通过考虑非线性的 S 盒部分, 说明当 n=128 时, 寄存器最少是 n/16=8 个. 此时 X_0 和 X_1 恰好各占用 4 个寄存器, 满足 4 比特 S 盒的计算要求, 再减少则会对 S 盒的计算造成影响.

(3) 通过考虑线性变换 B 部分, 说明将“竖”着的元素“横”着摆放是可行的.

(4) 通过考虑线性变换 PL/PR 部分, 给出了数据比特位在寄存器中存储的约束条件.

图 5 是本文针对 uBlock 算法的比特切片优化思想. 对于 n 比特分组长度的数据 X, 把 X 拆分成 n/2 比特长度的 X_0 和 X_1 分别进行处理; 以 X_0 为例, 将 n/2 比特的列向量看作是 1×n/2、2×n/4、4×n/8、8×n/16、16×n/32 的表示; 将“竖”着的元素“横”着摆放, 可以得到 n/2、n/4、n/8、n/16、n/32 行的数据; 每一行表示一个寄存器, 所以 X 得到需要 n、n/2、n/4、n/8 和 n/16 个寄存器的比特切片表示法.

本文提出的比特切片优化方法 FBS-uBlock 是灵活的, 可以从方法和应用这两个层面体现.

(1) 方法层面, 本文提出的 uBlock 算法比特切片优化方法在寄存器的占用数量上是灵活可调的. 本文并没有提出一种固定的比特切片表示法, 而是以一种减少寄存器占用数量的思想为引子, 构造了多种不同的 uBlock 算法比特切片表示法. 核心思想是在线性变换 B 中对数据矩阵重新表示, 由一维的列向量转变成二维的矩阵, 由“线”转变至“面”, 减少行数, 进而减少寄存器数量. 借助这个思想, 在不同的场景需求下, 可以灵活构造出占用不同寄存器数量的比特切片表示法.

(2) 应用层面, 本文提出的 uBlock 算法比特切片优化方法是一系列针对 uBlock 算法的比特切片表示法, 能够灵活适用于不同的处理器平台. 以 ARM 架构的处理器平台为例, 低端的处理器只有 16 个供 NEON 指令集操作的

向量寄存器, 高端的有 32 个向量寄存器, 未来说不定会产生具有 64 个向量寄存器的处理器, 开发者可以根据处理器平台的寄存器资源灵活选取最适合的比特切片表示法, 便利了 uBlock 算法的工程应用.

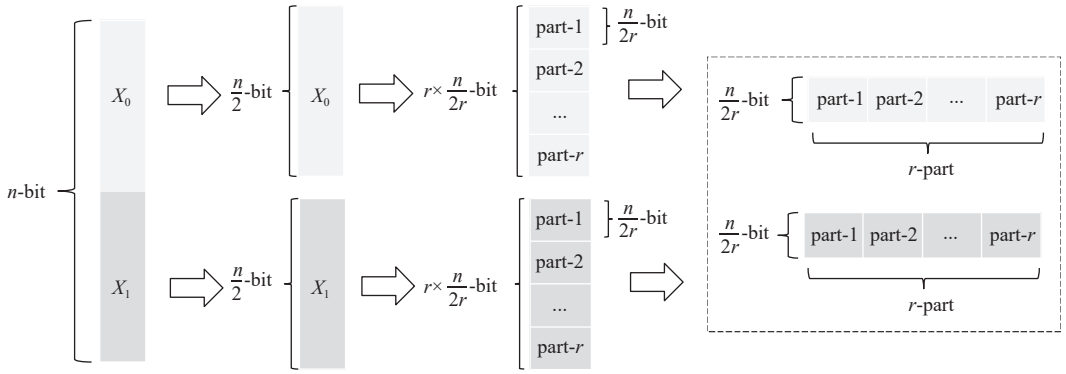

图 5 FBS-uBlock 核心思想

uBlock 算法的比特切片表示法 FBS-uBlock 的详细描述如图 6 所示, 图 6 涉及的符号定义如表 4 所示. 图 6 展示了 n 比特数据 X 在不同缩减比率 r 下的比特切片表示法, 占用的寄存器数量为 n/r 个, 需要一次性读入 $n/(2r)$ 组数据 X. 对于 uBlock-128/128 算法和 uBlock-128/256 算法, r 的取值为 1、2、4、8、16; 对于 uBlock-256/256 算法, r 还可以取到 32.

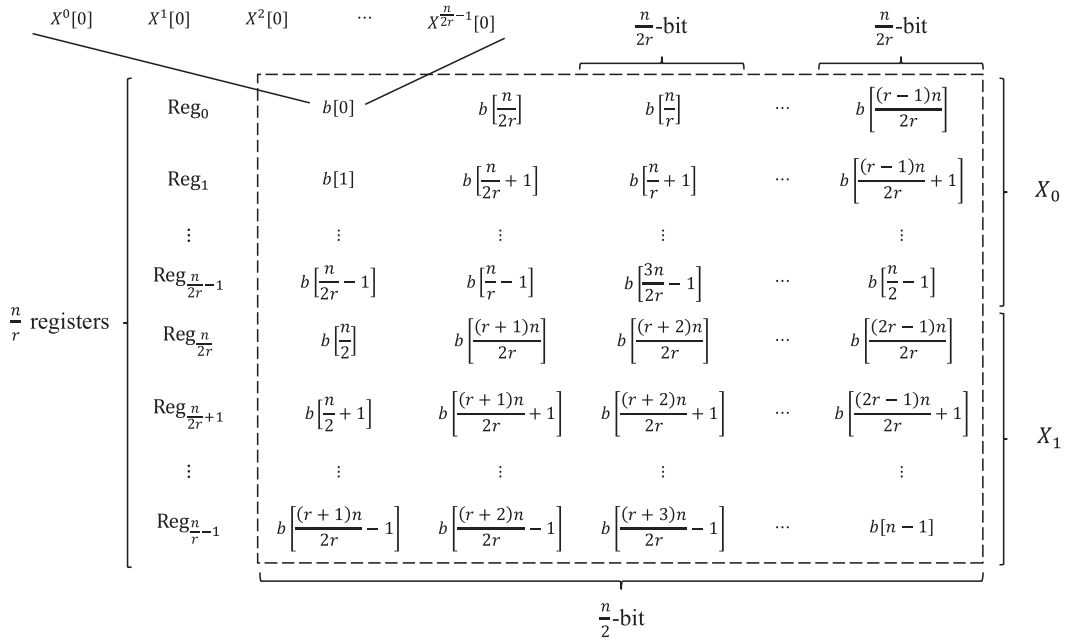

图 6 uBlock 算法比特切片表示法

图 6 中, $n/(2r)$ 组 n 比特数据 X 被转换成 $n/r \times n/2$ 大小的矩阵, 矩阵上半部分代表 X 的高 $n/2$ 比特, 下半部分代表 X 的低 $n/2$ 比特. 矩阵的上半或下半部分可以看成 $n/(2r) \times r$ 的分块矩阵, 每个分块元素代表 X 的某个比特位. X 的比特位将按照列的顺序放置, 从左上角开始, 由上至下, 由左至右. 每个分块元素 $b[i]$ 代表 $n/(2r)$ 组数据 X 的第 i 比特位, 长度恰好是 $n/(2r)$ 比特.

表 4 符号定义

符号	符号解释
X	uBlock算法的数据
n	数据X的比特长度, 代表X是n比特数据
X^i	第i组输入的数据
$X[j]$	数据X的第j比特
r	寄存器缩减率, 代表将寄存器占用个数减少到原先的1/r
$b[i]$	数据第i比特的集合, 长度$n/(2r)$, $b[i]=X^0[i]\ X^1[i]\ \dots\ X^{k-1}[i]$, $k=n/(2r)$

图 6 中矩阵共 $n/2$ 列, 但并不代表只适合 $n/2$ 比特长度的寄存器. 该切片表示法可以通过水平拼接矩阵的方式拓展到任意比特长度的寄存器中, 处理的数据量也等比例翻倍. 处理器的 SIMD 指令集支持并行的向量运算, 拼接后能够更好地发挥处理器的并行能力.

3.3 非线性 S 盒表达式

uBlock 算法设计文档中的 S 盒逻辑函数需要 8 个硬件逻辑门, 但在比特切片软件实现中, 处理器没有提供或非、与非等复合逻辑门的逻辑运算指令, 需要把或非门拆分成或门和非门这两步运算, 所以在软件比特切片中, 图 2 的逻辑函数需要耗费 14 条逻辑运算指令, 比特切片逻辑门复杂度为 14.

为了降低 S 盒的计算开销, 本文采用 Lighter 工具[15]搜索比特切片逻辑门复杂度更低的 S 盒逻辑表达式. Lighter 工具可以指定搜索时使用的逻辑门, 还能够设置各逻辑门的开销权重. 在 x86 架构的 AVX2 指令集中, 支持的逻辑运算是 and、or、xor、andn、not, 开销相同; 在 ARM 架构的 NEON 指令集中, 支持的逻辑运算是 and、or、xor、orn、not, 开销相同. 因此, 本文分别针对 x86 架构和 ARM 架构这两个处理器平台架构进行 S 盒逻辑函数的搜索与化简, 限制最终的逻辑函数仅能使用特定处理器平台支持的逻辑运算, 以减少比特切片门复杂度为目标搜索适合比特切片实现的 uBlock 算法 S 盒逻辑函数.

表 5 是 Lighter 工具搜索的结果, 图 7 是对应的逻辑函数. 根据搜索结果, 经过 Lighter 工具优化后的 S 盒比特切片门复杂度为 9, S 盒的逆也为 9. 并且搜索得到的逻辑函数不包含 andn 或 orn 逻辑运算, 这意味着同样的表达式可以同时用于 x86 和 ARM 架构两个平台.

表 5 比特切片逻辑门复杂度 (个)

模块	设计文档	x86-AVX2	ARM-NEON
$s(x)$	14	9	9
$s^{-1}(x)$	—	9	9

图 8 是 uBlock 算法 S 盒与逆 S 盒的实现方式, 式中对 4 比特的 X 执行多次逻辑运算来更新每个比特位的值, 执行完毕后的 X 是算法 S 盒或逆 S 盒的输出, 计算过程只需要使用 1 个额外的临时变量.

4 具体实现

本节主要介绍如何利用处理器的 SIMD 指令集实现本文提出的 FBS-uBlock, 依次介绍数据编排、轮密钥加、非线性 S 盒、线性变换 B、线性变换 PL 与 PR 这 5 部分的实现方法, 并理论分析需要的运算指令条数.

4.1 实现过程

数据编排指将原始输入数据转变成适合比特切片运算的数据的过程, 数据编排的逆过程称为数据逆编排. 本文的数据编排和数据逆编排算法在使用 AVX2 指令集实现时的指令开销都是 $2n/r+8n/r\times\log_2[n/(2r)]$ 条, 包括数据打包和数据切片两个子算法. 在数据打包算法中, 输入的 n 比特数据原本位于一个寄存器中, 需要拆分成高 $n/2$ 比特和低 $n/2$ 比特, 并分别统一放在两个不同的寄存器中. 在数据切片算法中, 采用第 2.2 节提到的数据转置算法, 完

成 2 个 $n/(2r)\times n/(2r)$ 的比特数据矩阵转置. 例如, 在数据打包算法中, 对于明文分组长度 n=128 比特的 uBlock-128/128 和 uBlock-128/256 算法, 如果寄存器长度是 128 比特, 那么每个寄存器中存放 2 个分组的高 (或低) 64 比特; 如果寄存器长度是 256 比特, 那么每个寄存器存放 4 个分组的高 (或低) 64 比特. 数据打包算法在 x86 平台可使用 unpack 指令快速完成. 对于 uBlock-256/256 算法, 如果寄存器长度恰好是 $n/2$=128 比特, 直接将数据加载至寄存器; 如果寄存器长度大于 128 比特, 利用 permute 指令可以快速完成数据打包.

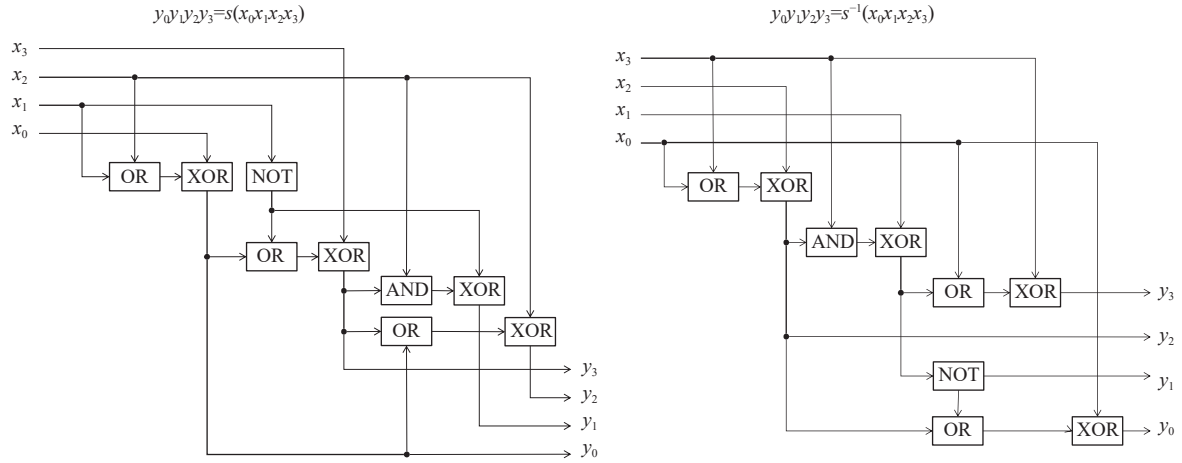

图 7　优化后的 uBlock 算法 $s(x)$ 和 $s^{-1}(x)$ 的逻辑门

S盒

```
X[0]=XOR2 (X[0], OR2 (X[1], X[2]));
X[1]=NOT1 (X[1]);
X[3]=XOR2 (X[3], OR2 (X[0], X[1]));
X[1]=XOR2 (X[1], AND2 (X[2], X[3]));
X[2]=XOR2 (X[2], OR2 (X[0], X[3]));
```

逆S盒

```
X[2]=XOR2 (X[2], OR2 (X[0], X[3]));
X[1]=XOR2 (X[1], AND2 (X[2], X[3]));
X[3]=XOR2 (X[3], OR2 (X[0], X[1]));
X[1]=NOT1 (X[1]);
X[0]=XOR2 (X[0], OR2 (X[1], X[2]));
```

图 8　uBlock 算法 $s(x)$ 和 $s^{-1}(x)$ 的实现方式

轮密钥加是轮密钥和数据的异或运算. 因为比特切片对输入的数据进行了编排, 所以轮密钥也需要进行类似的编排, 可以预先对每一轮的轮密钥做数据编排, 然后在加密和解密轮函数中直接用数据异或上编排好的轮密钥, 避免对密钥反复编排造成的开销.

非线性 S 盒使用第 3.2 节中给出的逻辑函数表达式计算, x86 架构的 SIMD 指令集不支持逻辑非运算指令, 实现时通过和全 1 的数据异或来实现逻辑非运算.

线性变换 B 是以 32 比特的循环移位和异或, 根据 $n/(2r)$ 的大小可以分为两种情况: 当 $n/(2r)\geqslant 32$ 时, 此时 32 比特都分散在不同寄存器中, 直接使用逻辑异或指令计算; 当 $n/(2r)<32$ 时, 利用第 3.1 节提到的性质, 使用重排指令快速循环移位, 完成线性变换 B 的计算.

线性变换 PL 和 PR 可以使用移位、掩码和重排指令实现, 计算开销随着寄存器的减少呈现先增加后降低的趋势. PL 和 PR 模块是规律性不强的字节重排, 本文为 uBlock 算法设计的比特切片表示法并不是非常切合 PL 和 PR 的计算, 无法仅使用简单的指令来抵消缩减寄存器数量带来的影响, 会引发寄存器和寄存器之间频繁的数据移动. 但通过后续的分析可以看出, 当寄存器数量减少到一定程度时, PL 和 PR 不再有寄存器与寄存器之间的交互, 可以使用重排指令快速完成字节重排.

4.2　理论分析

根据 Intel 官方指令集的文档 (Intel intrinsics guide), 逻辑运算指令、重排指令、打包指令的开销基本一致, 可

以仅统计这些运算指令的总条数来评估计算开销, 不需要单独统计每种指令. 下面以采用 AVX2 指令集实现的版本为例, 统计了 uBlock 算法加密过程中每个模块需要的运算指令条数和占比, 包括数据编排、轮密钥加、S 盒、线性变换 B、PL/PR 这 5 个模块.

表 6–表 8 统计了 uBlock 算法每个模块在不同切片表示法下平均每分组需要的运算指令条数, 用于分析寄存器数量减少对加密计算复杂度的影响, 每分组需要的运算指令条数通过“单次调用需要的运算指令条数/单次加密处理的分组数”来计算. 通过理论分析可以看出, 随着需要的寄存器数量减少 (r 增加), 轮密钥加和 S 盒的开销不变, 数据编排部分开销降低, 线性变换 B 的开销先不变后增加, PL/PR 模块开销先增高后降低. 从总共的运算指令条数上看, 趋势是先增高, 然后降低, 然后再升高.

表 6　uBlock-128/128 每个模块在不同比特切片表示下平均每分组需要的运算指令条数

表示法	数据编排	轮密钥加	S盒	线性变换B	PL/PR模块	总计
r=1	13	0.5	1.125	1.5	0	76.5
r=2	11	0.5	1.125	1.5	1.5	96.5
r=4	9	0.5	1.125	2	1.5	100.5
r=8	7	0.5	1.125	2.375	0.5	86.5
r=16	5	0.5	1.125	3.5	0.5	100.5

注: uBlock-128/128单次加密需要分别调用上述模块2、17、16、16、16次

表 7　uBlock-128/256 每个模块在不同比特切片表示下平均每分组需要的运算指令条数

表示法	数据编排	轮密钥加	S盒	线性变换B	PL/PR模块	总计
r=1	13	0.5	1.125	1.5	0	101.5
r=2	11	0.5	1.125	1.5	1.5	133.5
r=4	9	0.5	1.125	2	1.5	141.5
r=8	7	0.5	1.125	2.375	0.5	122.5
r=16	5	0.5	1.125	3.5	0.5	145.5

注: uBlock-128/256单次加密需要分别调用上述模块2、25、24、24、24次

表 8　uBlock-256/256 每个模块在不同比特切片表示下平均每分组需要的运算指令条数

表示法	数据编排	轮密钥加	S盒	线性变换B	PL/PR模块	总计
r=1	30	1	2.25	3	0	211.0
r=2	26	1	2.25	3	3	275.0
r=4	22	1	2.25	3	9.5	423.0
r=8	18	1	2.25	4	3	283.0
r=16	14	1	2.25	4.75	1	245.0
r=32	10	1	2.25	7	1	291.0

注: uBlock-256/256单次加密需要分别调用上述模块2、25、24、24、24次

表 9 给出了 PL/PR 模块需要的指令条数相对于整个加密过程的占比, 用于分析制约性能的主要模块, 模块需要的指令条数占比通过“该模块需要的指令条数/单次加解密需要的指令条数”来计算. 通过表 9 的数据可知, PL/PR 模块的开销随着寄存器数量的减少会急剧增大, 当 r=4 时, uBlock-256/256 算法 50% 的运算指令都用于计算该模块, 说明此时 PL/PR 模块制约着整个算法的性能. 但寄存器数量减少到一个临界值后, 运算指令占比便会急剧降低, uBlock-128/128 和 uBlock-128/256 的临界值是 r=8, 即 128/8=16 个寄存器; uBlock-256/256 的临界值是 r=16, 也是 256/16=16 个寄存器.

本文提出的切片优化方法关键在于减少寄存器数量, 间接减少了访存指令的条数, 但会增加一些运算指令的条数. 第 5 节实验中将会说明, 本文的 FBS-uBlock 方法有效地减少了 uBlock 算法比特切片优化时的访存指令条数.

表 9　PL/PR 模块相对于整个加密过程的运算指令条数占比 (%)

算法	r=1	r=2	r=4	r=8	r=16	r=32
uBlock-128/128	0	24.8	23.8	9.2	7.9	—
uBlock-128/256	0	26.9	25.4	9.7	8.2	—
uBlock-256/256	0	26.1	53.9	25.4	9.7	8.2

5　测试和分析

为了探究本文提出的 FBS-uBlock 优化方法在速率、访存上的效果, 本文在 x86 平台基于 AVX2 指令集使用 FBS-uBlock 优化方法分别实现了 uBlock-128/128、uBlock-128/256 和 uBlock-256/256, 并完成了性能测试和访存指令条数的统计分析. 测试使用的处理器平台是 Intel(R) Core(TM) i9-10900X, 基准频率 3.70 GHz, 最大睿频频率 4.50 GHz. 操作系统是 Ubuntu-20.04.2, 编译器选择 gcc 9.4.0, 开启编译器的 O3 优化.

5.1　性能测试

首先, 本文测试了 uBlock 算法在不同比特切片表示法下加密相同长度数据的速率, 数据大小是 16 384 字节, 使用相同的密钥循环加密 10 万次, 测试结果如表 10 所示, 对应的折线图如图 9 所示. 算法的参数 r 越大, 寄存器的数量越少. 从表 10 中可以看出, uBlock 算法的加密性能随寄存器数量的减少遵循先降低后增高的趋势. 对于 uBlock-128/128 和 uBlock-128/256 算法, 在 r=8 (或 r=16) 时的速率较高; 对于 uBlock-256/256 算法, 在 r=16 (或 r=32) 的速率较高. 此时需要的寄存器个数恰好都是 n/r=16 个 (或 8 个).

表 10　uBlock 算法不同切片表示下的加密性能测试 (Mb/s)

算法	r=1	r=2	r=4	r=8	r=16	r=32
uBlock-128/128	6 855	6 189	9 282	12 612	12 758	—
uBlock-128/256	5 059	4 475	6 583	8 944	8 834	—
uBlock-256/256	4 468	3 952	3 571	6 437	8 984	8 811

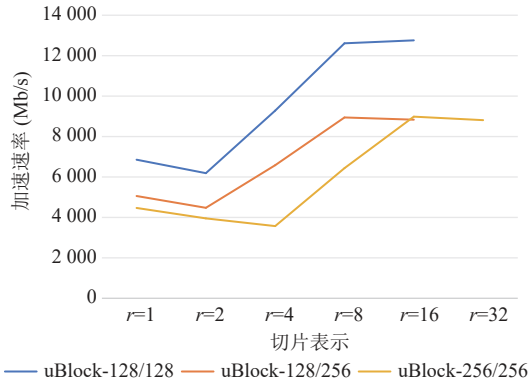

图 9　uBlock 算法不同切片表示下的加密性能

接着, 本文测试了 uBlock 算法在不同切片表示法下不同工作模式的速率, 分别测试了 ECB、CTR 和 CBC 工作模式, 加密的数据大小是 16 384 字节, 使用相同的密钥循环加密/解密 10 万次. 测试结果如表 11 所示.

对于 uBlock-128/128 算法和 uBlock-128/256 算法, 选择表 10 中性能表现较好的 r=8 和 r=16 的切片表示法进行测试; 对于 uBlock-256/256 算法, 选择表 10 中性能表现较好的 r=16 和 r=32 的切片表示法进行测试. 由表 11 可知, 本文的比特切片优化方法和设计文档中的实现相比有明显优势, ECB 和 CBC (解密) 模式下提升 4 倍, CTR 模式因为存在计数器自增模块所以提升幅度只有 2–3 倍.

表 11 uBlock 算法不同工作模式测试 (Mb/s)

工作模式	uBlock-128/128			uBlock-128/256			uBlock-256/256		
	设计文档	r=8	r=16	设计文档	r=8	r=16	设计文档	r=16	r=32
ECB (加密)	2573	12612	12758	1704	8944	8834	2037	8984	8811
ECB (解密)	2571	12627	12739	1717	8937	8937	2032	9266	8812
CTR (加密)	2242	8164	8176	1545	6453	6404	1969	6887	6733
CTR (解密)	2411	8149	8172	1638	6421	6315	1983	6927	6726
CBC (解密)	2510	12276	12378	1678	8715	8576	2033	8643	8669

5.2 访存测试

本文统计了 uBlock 算法在不同比特切片表示法下加密数据时平均每分组需要的访存指令的条数, 将涉及到内存的指令都视作访存指令, 通过“单次加密需要的访存指令条数/单次加密处理的分组”计算, 统计结果如表 12 所示. 由表 12 可知, 在 r 相同的情况下, uBlock-128/128、uBlock-128/256 和 uBlock-256/256 每分组平均需要的访存指令条数依次增加; 对于同一种 uBlock 算法, 随着 r 的增加, 访存指令条数在 r=2 时会增高, 在 r>2 后才会呈现下降的趋势.

表 12 uBlock 算法不同比特切片表示下平均每分组需要的访存指令的条数

算法	r=1	r=2	r=4	r=8	r=16	r=32
uBlock-128/128	114.6	123.8	99.8	55.0	32.8	—
uBlock-128/256	161.3	177.2	134.9	78.5	47.3	—
uBlock-256/256	340.1	394.4	389.2	292.6	153.8	94.3

本文计算了 uBlock 算法在不同比特切片表示法下加密数据时平均访存指令条数同基准的比值, 基准是 Biham 的比特切片表示法 (r=1), 统计结果如表 13 所示. 由表 13 可知, 随着寄存器数量的减少, 3 种版本的 uBlock 算法的平均访存指令都遵循先增高后降低的趋势, 最多能够分别降低 71%、71% 和 72%. 相比于 r=1 时的比特切片表示法, uBlock-128/128 和 uBlock-128/256 算法在 r≥4 时才会有访存的降低, uBlock-256/256 算法在 r≥8 时才会有访存的降低, 产生这种现象的原因有两个: (1) r>1 时的比特切片表示法比 r=1 时的表示法需要更多运算指令, 间接使得读取运算操作数和存储运算结果的访存指令条数增加; (2) 当 r 没有达到一个临界值时, 算法需要的寄存器数量仍远大于处理器拥有的寄存器数量, 大部分的运算结果仍然无法驻留在寄存器中, 下一次用到这些数据时又需要耗费访存指令.

表 13 uBlock 算法不同比特切片表示法下加密数据时平均访存指令条数同基准的比值

算法	r=1	r=2	r=4	r=8	r=16	r=32
uBlock-128/128	1	1.080	0.870	0.479	0.285	—
uBlock-128/256	1	1.098	0.892	0.486	0.292	—
uBlock-256/256	1	1.159	1.144	0.860	0.452	0.277

本文计算了 uBlock 算法在不同比特切片表示法下访存指令占比, 运算指令使用第 4.2 节中表 6–表 8 的计算结果, 统计结果如表 14 所示. 根据表 14 的测试数据可知, 随着寄存器个数的减少, 访存指令占全部指令的比例也在减少, 从最初的 60% 左右降低到 25% 左右. uBlock-256/256 算法的访存指令占比随寄存器减少并不是单调递减, 因为当 r=4 至 r=8 时 PL/PR 的计算开销大幅降低, 反过来使得访存指令的占比增加.

表 14 uBlock 算法不同比特切片表示法下访存指令占比 (%)

算法	r=1	r=2	r=4	r=8	r=16	r=32
uBlock-128/128	60.0	56.2	49.8	38.9	24.6	—
uBlock-128/256	61.4	57.0	48.8	39.1	24.5	–
uBlock-256/256	61.7	58.9	47.9	50.8	38.6	24.5

5.3 性能对比分析

根据第 4.1 节中表 10 和表 11 的性能测试数据, 无论是 uBlock-128/128、uBlock-128/256 还是 uBlock-256/256, 都是当寄存器个数为 16 或 8 个时性能最优, 采用 16 个寄存器的优化方法要略优于 8 个寄存器的优化方法. 这说明在 x86 平台使用 AVX2 指令集对 uBlock 算法加速优化时, 选择 16 个寄存器的方案最优.

根据第 5.2 节的访存指令的统计数据和第 4.2 节的运算指令的统计数据可知, 如果需要的寄存器数量多, 会导致较多的访存开销 (如表 12 所示); 如果方法需要的寄存器数量少, 会导致较多的计算开销 (如表 6–表 8 所示). 实验结果说明, 采用 16 个寄存器的方法在访存消耗和计算消耗上达到平衡, 具有最优的性能, 这恰好是 AVX2 指令集提供的 YMM 寄存器个数. 如果需要将优化方法迁移到 ARM 平台, 本文的实验结果仍然具备一定的参考价值.

表 15 和表 16 是本文的优化方法与其他算法的对比. 为了减弱测试平台的差异所带来的影响, 本文使用 cpB (cycles per byte, 每字节需要的处理器时钟周期数) 指标与 Mbps (megabits per second, 每秒传输的兆比特数) 指标进行对比. 可以看出, 本文提出的 uBlock 优化方法在不同分组长度和工作模式下均表现出性能优势. 本文优化后的 uBlock 算法优于现有的 uBlock 官方实现[1]和公开优化方法[44]. 在 CBC 模式下 (表 15), 优化实现的解密速率比文献 [1] 的 uBlock 官方实现提升了约 3–4 倍; 在 ECB 模式下 (表 16), 优化实现的加密速率比文献 [44] 的 uBlock 优化实现提升约 1–2 倍. 与其他密码算法相比, 性能显著优于 ANT[48]、FESH[37]、TANGRAM[38]算法, 略优于 Ballet[49]、AES[30]和 SM4[32]算法, 在不同分组长度和工作模式下均具备优异的处理速度和较低的计算开销.

表 15　不同算法的 CBC 解密性能对比

算法	优化方法	128/128*		128/256*		256/256*		平台
		Mbps	cpB	Mbps	cpB	Mbps	cpB	
uBlock	本文	12276	2.35	8715	3.32	8643	3.34	i9-10900X @ 3.70 GHz
uBlock	文献[1]	1869	11.29	1289	16.36	1571	13.43	i7-3740QM @ 2.70 GHz
Ballet	文献[49]	7899	2.77	7080	3.09	4425	4.94	i7-6700T @ 2.80 GHz
ANT	文献[48]	6065	4.38	6137	4.33	2635	10.08	i7-6700 @ 3.40 GHz
FESH	文献[37]	4418	6.37	3664	7.68	3191	8.81	i7-4790QM @ 3.60 GHz

注: *为分组长度/密钥长度; uBlock算法的测试数据是文献[1]中对1 MB长消息加密的速度检测结果

表 16　不同算法的 ECB 加密性能对比

算法	优化方法	128/128*		128/256*		256/256*		平台
		Mbps	cpB	Mbps	cpB	Mbps	cpB	
uBlock	本文	12758	2.26	8944	3.23	8984	3.21	i9-10900X @ 3.70 GHz
uBlock	文献[44]	7205	4.01	4099	7.05	3182	9.08	AMD Ryzen 9 5900X @ 3.70 GHz
AES	文献[30]	9799	2.55	—	—	—	—	i7-8700H @ 3.20 GHz
SM4	文献[32]	7813	3.20	—	—	—	—	i7-8700 @ 3.20 GHz
TANGRAM	文献[38]	5461	4.86	4802	5.53	2963	8.96	i7-6700 @ 3.40 GHz

注: *为分组长度/密钥长度; AES算法的测试数据是文献[30]中采用AVX2指令集实现的版本

6 总　结

本文从数据的访存开销入手, 通过仔细分析 uBlock 算法结构, 给出了 uBlock 算法降低在比特切片下寄存器数量的理论依据, 为 uBlock 算法设计了一种灵活的比特切片优化方法 FBS-uBlock, 包括 uBlock 算法多种不同的比特切片表示法, 降低算法在比特切片下占用的寄存器数量, 进而降低访存开销. 实验结果表明, 优化后的 uBlock 算法比特切片实现的加密速率可达到 12758 Mb/s、8944 Mb/s、8984 Mb/s, 比设计文档提升了 3.9、4.2、3.4 倍. 此外, 根据实验分析, 通过增加少许额外的计算开销, 优化的 uBlock 算法的访存指令的条数明显降低, 最多能够分别降低 71%、71% 和 72%, 有效地减少了访存开销. 在工程实现上, 本文提出的 uBlock 算法优化方法还有进一步

优化的空间, 在指令集的使用上可以进一步优化. 本文在设计优化方法时选择了两个处理器平台 (x86 和 ARM) 都兼容的指令运算, 但这两个平台都有自己特有的指令. 例如, x86 平台的 AVX512 指令集有 3 操作数逻辑指令 vpternlogq, 可更快地计算逻辑函数; ARM 平台的 NEON 指令集支持数据交织存取的操作, 可更快地完成数据编排. 后续可以考虑基于本文提出的比特切片优化方法 FBS-uBlock, 针对某一具体处理器平台对 uBlock 算法进行进一步优化. 此外, 针对 CBC 加密、OFB 等具有分组间数据依赖的工作模式, Bogdanov 等人[50]提出了基于前瞻策略的调度器 Comb Scheduler, 利用多个独立消息流填充 AES-NI 指令集的指令流水. 后续可以考虑将本文的优化方法与 Comb Scheduler 结合, 加速 CBC 加密等具有分组间数据依赖的工作模式.

References:

[1] Wu WL, Zhang L, Zheng YF, Li LC. The block cipher uBlock. Journal of Cryptologic Research, 2019, 6(6): 690–703 (in Chinese with English abstract). [doi: 10.13868/j.cnki.jcr.000334]

[2] Li XD, Wu WL, Zhang L. Efficient search for optimal vector permutations of uBlock-like structures. Journal of Computer Research and Development, 2022, 59(10): 2275–2285 (in Chinese with English abstract). [doi: 10.7544/issn1000-1239.20220485]

[3] Yang YT, Dong H, Liu JT, Zhang YS. AEUR: Authenticated encryption algorithm design based on uBlock round function. Journal on Communications, 2023, 44(8): 168–178 (in Chinese with English abstract). [doi: 10.11959/j.issn.1000-436x.2023159]

[4] Jiao ZP, Chen H, Yao F, Fan LM. The low cost threshold implementation method of uBlock algorithm against side channel attacks. Chinese Journal of Computers, 2023, 46(3): 657–670 (in Chinese with English abstract). [doi: 10.11897/SP.J.1016.2023.00657]

[5] Liu CJ, Zhang YW, Xu JJ, Zhao J, Xiang SH. Ensuring the security and performance of IoT communication by improving encryption and decryption with the lightweight cipher uBlock. IEEE Systems Journal, 2022, 16(4): 5489–5500. [doi: 10.1109/JSYST.2022.3140850]

[6] Tian WQ, Hu B. Integral cryptanalysis on two block ciphers Pyjamask and uBlock. IET Information Security, 2020, 14(5): 572–579. [doi: 10.1049/iet-ifs.2019.0624]

[7] Wang QL, Lu JQ. Fault analysis of the ARIA and uBlock block ciphers. In: Proc. of the 2021 IEEE Int'l Conf. on Service Operations and Logistics, and Informatics (SOLI). Singapore: IEEE, 2021. 1–6. [doi: 10.1109/SOLI54607.2021.9672378]

[8] Zhang L, Zhang Y, Wu WL, Mao YX, Zheng YF. Explicit upper bound of impossible differentials for AES-like ciphers: Application to uBlock and Midori. The Computer Journal, 2024, 67(2): 674–687. [doi: 10.1093/comjnl/bxad009]

[9] Xin JY, Du ZB. Template attack based on uBlock cipher algorithm. Frontiers in Computing and Intelligent Systems, 2023, 3(1): 90–93. [doi: 10.54097/fcis.v3i1.6031]

[10] Huang M, Zhang SS, Hong CL, Zeng L, Xiang ZJ. MILP modeling of division property propagation for block ciphers with complex linear layers. Ruan Jian Xue Bao/Journal of Software, 2024, 35(4): 1980–1992 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6839.htm [doi: 10.13328/j.cnki.jos.006839]

[11] Biham E. A fast new DES implementation in software. In: Proc. of the 4th Int'l Workshop on Fast Software Encryption. Haifa: Springer, 1997. 260–272. [doi: 10.1007/BFb0052352]

[12] Könighofer R. A fast and cache-timing resistant implementation of the AES. In: Topics in Cryptology—CT-RSA 2008, The Cryptographer's Track at the RSA Conf. 2008. San Francisco: Springer, 2008. 187–202. [doi: 10.1007/978-3-540-79263-5_12]

[13] Käsper E, Schwabe P. Faster and timing-attack resistant AES-GCM. In: Proc. of the 11th Int'l Workshop Lausanne Cryptographic Hardware and Embedded Systems (CHES 2009). Springer, 2009. 1–17. [doi: 10.1007/978-3-642-04138-9_1]

[14] Chen C, Guo H, Liu YH, Gong ZR, Zhang YX. Optimization implementation method of SM4 based on register. Journal of Cryptologic Research, 2024, 11(2): 427–440 (in Chinese with English abstract). [doi: 10.13868/j.cnki.jcr.000686]

[15] Jean J, Peyrin T, Sim SM, Tourteaux J. Optimizing implementations of lightweight building blocks. IACR Trans. on Symmetric Cryptology, 2017, 2017(4): 130–168. [doi: 10.13154/tosc.v2017.i4.130-168]

[16] Kwan M. Reducing the gate count of bitslice DES. Cryptology ePrint Archive, 2000/051, 2000.

[17] May L, Penna L, Clark A. An implementation of bitsliced DES on the pentium MMX^{TM} processor. In: Proc. of the 5th Australasian Conf. on Information Security and Privacy. Brisbane: Springer, 2000. 112–122. [doi: 10.1007/10718964_10]

[18] Schaumüller-Bichl I. Cryptanalysis of the data encryption standard by the method of formal coding. In: Proc. of the 1983 Workshop on Cryptography. Burg Feuerstein: Springer, 1983. 235–255. [doi: 10.1007/3-540-39466-4_17]

[19] Matsuda S, Moriai S. Lightweight cryptography for the cloud: Exploit the power of bitslice implementation. In: Proc. of the 14th Int'l Workshop on Cryptographic Hardware and Embedded Systems (CHES 2012). Leuven: Springer, 2012. 408–425. [doi: 10.1007/978-3-642-33027-8_24]

[20] Papapagiannopoulos K. High throughput in slices: The case of PRESENT, PRINCE and KATAN64 ciphers. In: Proc. of the 10th Int'l Workshop on Radio Frequency Identification: Security and Privacy Issues. Oxford: Springer, 2014. 137–155. [doi: 10.1007/978-3-319-13066-8_9]

[21] Zhang XC, Guo H, Zhang XY, Wang C, Liu JW. Fast software implementation of SM4. Journal of Cryptologic Research, 2020, 7(6): 799–811 (in Chinese with English abstract). [doi: 10.13868/j.cnki.jcr.000407]

[22] Wang L, Gong Z, Liu Z, Chen JH, Hao JF. Fast software implementation of SM4 based on tower field. Journal of Cryptologic Research, 2022, 9(6): 1081–1098 (in Chinese with English abstract). [doi: 10.13868/j.cnki.jcr.000569]

[23] Chen C, Guo H, Wang C, Liu YH, Liu JW. A fast software implementation of SM4 based on composite fields. Journal of Cryptologic Research, 2023, 10(2): 289–305 (in Chinese with English abstract). [doi: 10.13868/j.cnki.jcr.000594]

[24] Rudra A, Dubey PK, Jutla CS, Kumar V, Rao JR, Rohatgi P. Efficient Rijndael encryption implementation with composite field arithmetic. In: Proc. of the 3rd Int'l Workshop on Cryptographic Hardware and Embedded Systems (CHES 2001). Paris: Springer, 2001. 171–184. [doi: 10.1007/3-540-44709-1_16]

[25] Rebeiro C, Selvakumar D, Devi ASL. Bitslice implementation of AES. In: Proc. of the 5th Int'l Conf. on Cryptology and Network Security. Suzhou: Springer, 2006. 203–212. [doi: 10.1007/11935070_14]

[26] Matsui M, Nakajima J. On the power of Bitslice implementation on Intel Core2 processor. In: Proc. of the 9th Int'l Workshop on Cryptographic Hardware and Embedded Systems - CHES 2007. Vienna: Springer, 2007. 121–134. [doi: 10.1007/978-3-540-74735-2_9]

[27] Schwabe P, Stoffelen K. All the AES you need on Cortex-M3 and M4. In: Proc. of the 23rd Int'l Conf. on Selected Areas in Cryptography (SAC 2016). St. John's: Springer, 2017. 180–194. [doi: 10.1007/978-3-319-69453-5_10]

[28] Xu RQ, Xiang ZJ, Lin D, Zhang SS, He DB, Zeng XY. High-throughput block cipher implementations with SIMD. Journal of Information Security and Applications, 2022, 70: 103333. [doi: 10.1016/j.jisa.2022.103333]

[29] Nishikawa N, Amano H, Iwai K. Implementation of Bitsliced AES encryption on CUDA-enabled GPU. In: Proc. of the 11th Int'l Conf. on Network and System Security. Helsinki: Springer, 2017. 273–287. [doi: 10.1007/978-3-319-64701-2_20]

[30] Hajihassani O, Monfared SK, Khasteh SH, Gorgin S. Fast AES implementation: A high-throughput bitsliced approach. IEEE Trans. on Parallel and Distributed Systems, 2019, 30(10): 2211–2222. [doi: 10.1109/TPDS.2019.2911278]

[31] Miao X, Guo C, Wang MQ, Wang WJ. How fast can SM4 be in software? In: Proc. of the 18th Int'l Conf. on Information Security and Cryptology. Beijing: Springer, 2023. 3–22. [doi: 10.1007/978-3-031-26553-2_1]

[32] Miao X. Fast software implementations of SM4 [MS. Thesis]. Ji'nan: Shandong University, 2023 (in Chinese with English abstract). [doi: 10.27272/d.cnki.gshdu.2023.003953]

[33] Wang C, Ding Y, Huang CL, Song LT. Bitsliced optimization of SM4 algorithm with the SIMD instruction set. Journal of Computer Research and Development, 2024, 61(8): 2097–2109 (in Chinese with English abstract). [doi: 10.7544/issn1000-1239.202220531]

[34] Adomnicai A, Peyrin T. Fixslicing AES-like ciphers: New bitsliced AES speed records on ARM-cortex M and RISC-V. IACR Trans. on Cryptographic Hardware and Embedded Systems, 2021, 2021(1): 402–425. [doi: 10.46586/tches.v2021.i1.402-425]

[35] Banik S, Pandey SK, Peyrin T, Sasaki Y, Sim SM, Todo Y. GIFT: A small present: Towards reaching the limit of lightweight encryption. In: Proc. of the 19th Int'l Conf. on Cryptographic Hardware and Embedded Systems (CHES 2017). Taipei: Springer, 2017. 321–345. [doi: 10.1007/978-3-319-66787-4_16]

[36] Dobraunig C, Eichlseder M, Mendel F, Schläffer M. ASCON v1.2: Lightweight authenticated encryption and hashing. Journal of Cryptology, 2021, 34(3): 33. [doi: 10.1007/s00145-021-09398-9]

[37] Jia KT, Dong XY, Wei CM, Li Z, Zhou HB, Cong TS. On the design of block cipher FESH. Journal of Cryptologic Research, 2019, 6(6): 713–726 (in Chinese with English abstract). [doi: 10.13868/j.cnki.jcr.000336]

[38] Zhang WT, Ji FL, Ding TY, Yang BH, Zhao XF, Xiang ZJ, Bao ZZ, Liu LB. TANGRAM: A bit-slice block cipher suitable for multiple platforms. Journal of Cryptologic Research, 2019, 6(6): 727–747 (in Chinese with English abstract). [doi: 10.13868/j.cnki.jcr.000337]

[39] Reis TBS, Aranha DF, López J. PRESENT runs fast: Efficient and secure implementation in software. In: Proc. of the 19th Int'l Conf. on Cryptographic Hardware and Embedded Systems (CHES 2017). Taipei: Springer, 2017. 644–664. [doi: 10.1007/978-3-319-66787-4_31]

[40] Hamburg M. Accelerating AES with vector permute instructions. In: Proc. of the 11th Int'l Workshop Lausanne Cryptographic Hardware and Embedded Systems (CHES 2009). Springer, 2009. 18–32. [doi: 10.1007/978-3-642-04138-9_2]

[41] Seo H, Park T, Heo S, Seo G, Bae B, Hu Z, Zhou L, Nogami Y, Zhu YW, Kim H. Parallel implementations of LEA, revisited. In: Proc. of the 17th Int'l Workshop on Information Security Applications. Jeju Island: Springer, 2017. 318–330. [doi: 10.1007/978-3-319-56549-1_27]

[42] Park T, Seo H, Lee G, Khandaker MAA, Nogami Y, Kim H. Parallel implementations of SIMON and SPECK, revisited. In: Proc. of the 18th Int'l Conf. on Information Security Applications. Jeju Island: Springer, 2018. 283–294. [doi: 10.1007/978-3-319-93563-8_24]

[43] Adomnicai A, Minematsu K, Shigeri M. Fast Skinny-128 SIMD implementations for sequential modes of operation. In: Proc. of the 27th Australasian Conf. on Information Security and Privacy. Wollongong: Springer, 2022. 125–144. [doi: 10.1007/978-3-031-22301-3_7]

[44] Gao Y, Wang LX, Tian L, Hu Y, Zhang YP, Yan Y, Wu QH. Fast software implementation of the block cipher uBlock algorithm. Journal of National University of Defense Technology, 2024, 46(6): 96–106 (in Chinese with English abstract). [doi: 10.11887/j.cn.202406010]

[45] Hutter M, Wenger E. Fast multi-precision multiplication for public-key cryptography on embedded microprocessors. Journal of Cryptology, 2020, 33(4): 1442–1460. [doi: 10.1007/s00145-020-09351-2]

[46] Hutter M, Schwabe P. Multiprecision multiplication on AVR revisited. Journal of Cryptographic Engineering, 2015, 5(3): 201–214. [doi: 10.1007/s13389-015-0093-2]

[47] Seo H, Kim H. Multi-precision multiplication for public-key cryptography on embedded microprocessors. In: Proc. of the 13th Int'l Workshop on Information Security Applications. Jeju Island: Springer, 2012. 55–67. [doi: 10.1007/978-3-642-35416-8_5]

[48] Chen SY, Fan YH, Fu Y, Huang LN, Wang MQ. On the design of ANT family block ciphers. Journal of Cryptologic Research, 2019, 6(6): 748–759 (in Chinese with English abstract). [doi: 10.13868/j.cnki.jcr.000338]

[49] Cui TT, Wang MQ, Fan YH, Hu K, Fu Y, Huang LN. Ballet: A software-friendly block cipher. Journal of Cryptologic Research, 2019, 6(6): 704–712 (in Chinese with English abstract). [doi: 10.13868/j.cnki.jcr.000335]

[50] Bogdanov A, Lauridsen MM, Tischhauser E. Comb to pipeline: Fast software encryption revisited. In: Proc. of the 22nd Int'l Workshop on Fast Software Encryption. Istanbul: Springer, 2015. 150–171. [doi: 10.1007/978-3-662-48116-5_8]

附中文参考文献:

[1] 吴文玲, 张蕾, 郑雅菲, 李灵琛. 分组密码 uBlock. 密码学报, 2019, 6(6): 690–703. [doi: 10.13868/j.cnki.jcr.000334]

[2] 李晓丹, 吴文玲, 张丽. uBlock 类结构最优向量置换的高效搜索. 计算机研究与发展, 2022, 59(10): 2275–2285. [doi: 10.7544/issn1000-1239.20220485]

[3] 杨亚涛, 董辉, 刘建韬, 张艳硕. AEUR: 基于 uBlock 轮函数的认证加密算法设计. 通信学报, 2023, 44(8): 168–178. [doi: 10.11959/j.issn.1000-436x.2023159]

[4] 焦志鹏, 陈华, 姚富, 范丽敏. uBlock 算法的低代价门限实现侧信道防护方法. 计算机学报, 2023, 46(3): 657–670. [doi: 10.11897/SP.J.1016.2023.00657]

[10] 黄明, 张莎莎, 洪春雷, 曾乐, 向泽军. 分组密码复杂线性层可分性传播的 MILP 刻画方法. 软件学报, 2024, 35(4): 1980–1992. http://www.jos.org.cn/1000-9825/6839.htm [doi: 10.13328/j.cnki.jos.006839]

[14] 陈晨, 郭华, 刘源灏, 龚子睿, 张宇轩. 基于寄存器的 SM4 软件优化实现方法. 密码学报, 2024, 11(2): 427–440. [doi: 10.13868/j.cnki.jcr.000686]

[21] 张笑从, 郭华, 张习勇, 王闯, 刘建伟. SM4 算法快速软件实现. 密码学报, 2020, 7(6): 799–811. [doi: 10.13868/j.cnki.jcr.000407]

[22] 王磊, 龚征, 刘哲, 陈锦海, 郝金福. 基于塔域的 SM4 算法快速软件实现. 密码学报, 2022, 9(6): 1081–1098. [doi: 10.13868/j.cnki.jcr.000569]

[23] 陈晨, 郭华, 王闯, 刘源灏, 刘建伟. 一种基于复合域的国密 SM4 算法快速软件实现方法. 密码学报, 2023, 10(2): 289–305. [doi: 10.13868/j.cnki.jcr.000594]

[32] 苗鑫. SM4 分组密码算法的快速软件实现 [硕士学位论文]. 济南: 山东大学, 2023. [doi: 10.27272/d.cnki.gshdu.2023.003953]

[33] 王闯, 丁澔, 黄辰林, 宋连涛. 面向 SIMD 指令集的 SM4 算法比特切片优化. 计算机研究与发展, 2024, 61(8): 2097–2109. [doi: 10.7544/issn1000-1239.202220531]

[37] 贾珂婷, 董晓阳, 魏淙洺, 李铮, 周海波, 丛天硕. 分组密码算法 FESH. 密码学报, 2019, 6(6): 713–726. [doi: 10.13868/j.cnki.jcr.000336]

[38] 张文涛, 季福磊, 丁天佑, 杨博翰, 赵雪锋, 向泽军, 包珍珍, 刘雷波. TANGRAM: 一个基于比特切片的适合多平台的分组密码. 密码学报, 2019, 6(6): 727–747. [doi: 10.13868/j.cnki.jcr.000337]

[44] 高莹, 汪龙昕, 田蕾, 胡洋, 张宇鹏, 严宇, 伍前红. 分组密码 uBlock 算法快速软件实现. 国防科技大学学报, 2024, 46(6): 96–106. [doi: 10.11887/j.cn.202406010]

[48] 陈师尧, 樊燕红, 付勇, 黄鲁宁, 王美琴. ANT 系列分组密码算法. 密码学报, 2019, 6(6): 748–759. [doi: 10.13868/j.cnki.jcr.000338]

[49] 崔婷婷, 王美琴, 樊燕红, 胡凯, 付勇, 黄鲁宁. Ballet: 一个软件实现友好的分组密码算法. 密码学报, 2019, 6(6): 704–712. [doi: 10.13868/j.cnki.jcr.000335]

龚子睿(2001－), 男, 硕士生, 主要研究领域为密码学.

张宇轩(2000－), 男, 硕士生, 主要研究领域为密码学.

郭华(1980－), 女, 博士, 副教授, CCF 专业会员, 主要研究领域为密码学.

陈俊鑫(2000－), 男, 硕士生, 主要研究领域为密码学.

陈晨(1995－), 男, 博士生, 主要研究领域为密码学.

关振宇(1984－), 男, 博士, 教授, CCF 专业会员, 主要研究领域为密码工程.