doi:10.11887/j.cn.202406010 http://journal.nudt.edu.cn

分组密码 uBlock 算法快速软件实现

高 莹*,汪龙昕,田 蕾,胡 洋,张宇鹏,严 宇,伍前红
(北京航空航天大学 网络空间安全学院,北京 100191)

摘 要:为对国产分组密码算法 uBlock 进行软件优化,选择支持 256 bit 数据位宽的 AVX2 指令集,提高编译器自动优化等级,优化函数的调用过程,优化数据存储结构,综合使用高位并行、低延迟指令逻辑优化等方法实现单线程并行计算。通过使用这种有效的组合方法,uBlock-128/128 算法、uBlock-128/256 算法和 uBlock-256/256 算法单密钥短消息加密的速度较原代码分别提升 269%、182% 和 49%。基于这些优化方法,uBlock-128/128、uBlock-128/256 和 uBlock-256/256 三个算法版本均实现了单密钥场景与多密钥场景。

关键词:uBlock 算法;AVX2 指令集;并行运算;低延迟;快速软件实现
中图分类号:TP309.7 **文献标志码**:A **文章编号**:1001-2486(2024)06-096-11

论文拓展

Fast software implementation of the block cipher uBlock algorithm

*GAO Ying**, *WANG Longxin*, *TIAN Lei*, *HU Yang*, *ZHANG Yupeng*, *YAN Yu*, *WU Qianhong*
(School of Cyber Science and Technology, Beihang University, Beijing 100191, China)

Abstract: To optimize the software implementation of the domestic block cipher uBlock algorithm, the AVX2 instruction set supporting 256 bit data width was implemented, the automatic optimization level of the compiler was increased, optimizing the calling process of functions, and the methods of data storage structure optimization, high-level parallelism and low latency instruction logic optimization were used in order to implement parallel computing under the single-thread condition. Using this efficient combination method, the speed of single key short message encryption of uBlock-128/128 algorithm, uBlock-128/256 algorithm and uBlock-256/256 algorithm are 269%, 182% and 49% higher than the original code. Based on these optimization methods, the implementation of single-key scenario and multi-key scenario are given for three algorithm versions of uBlock-128/128, uBlock-128/256 and uBlock-256/256.

Keywords: uBlock algorithm; AVX2 instruction set; parallel operation; low latency; fast software implementation

2016 年,我国发布《国家网络空间安全战略》,其中明确指出我国网络安全正遭遇前所未有的挑战,也进一步确立了我国网络空间安全体系的建设目标和战略任务,同时也明确了国产密码算法在我国建设安全性更加完备的网络空间时代的重要地位。2020 年,美国政府相继发布《5G 安全国家战略》《关键与新兴技术国家战略》等重要文件。2021 年,欧盟公开了经过修改完善的网络安全相关文件——《欧盟数字十年的网络安全战略》[1]。由此可见,各国已经开始加紧研究,力求在网络空间中占据有利地位。与此同时,全球网络安全博弈频度上升,网络安全已成为全领域影响因素。

信息安全是网络空间安全领域的重要部分。在我国信息化进程加速的背景下,各行各业对信息处理速度和数据安全性的要求越来越高。而密码技术作为网络世界最基本、最核心的安全技术之一,在保证协议安全、通信安全、文件安全、实体认证、抗抵赖等方面都具有重要作用[2-4],其对国家网络安全的支撑作用也日益明显。因此,各国纷纷加紧密码技术的研究,并且制定政策对密码技术及加密产品进行出口管制[5]。其中,美国作为网络强国,掌握大量先进密码技术,向世界各国提供相关产品及服务,几近垄断。但

收稿日期:2022-04-06
基金项目:国家重点研发计划资助项目(2022YFB2701600);国家自然科学基金资助项目(61932011,61932011,61972017);北京市自然科学基金资助项目(M21033)
*第一作者:高莹(1977—),女,湖北大悟人,副教授,博士,博士生导师,E-mail:gaoying@buaa.edu.cn
引用格式:高莹,汪龙昕,田蕾,等. 分组密码 uBlock 算法快速软件实现[J]. 国防科技大学学报,2024,46(6):96-106.
Citation: GAO Y, WANG L X, TIAN L, et al. Fast software implementation of the block cipher uBlock algorithm [J]. Journal of National University of Defense Technology, 2024, 46(6): 96-106.

2013 年美国国家安全局旨在监视并窃取全球数据的“棱镜计划”被曝光,为世界各国敲响了警钟:拥有自主可控的技术才能从根本上保障国家信息安全[6]。

随着我国商用密码技术的应用推广,针对安全性高、可扩展性好、适应性强、可满足多个行业领域应用需求的国产密码算法受到了广泛关注。围绕这些优秀的国产密码算法,一个研究方向是评估它们的安全性,而另一个方向则是研究它们在各种应用环境下的软件优化实现技术。密码算法需要很高的软件实现性能,与面向硬件的设计相比,软件实现具有更好的灵活性和更低的实现成本。

密码算法的软件实现评估主要分为对软件速度的测试和对存储空间的测试,在计算资源十分受限的设备上进行密码算法实现时需要评估存储空间。因此,分组密码的软件快速实现有两个研究方向:一是轻量级分组密码算法的多平台软件实现[7];二是对现有的典型分组密码算法进行快速软件实现,多见于对国外的高级加密标准(advanced encryption standard, AES)等被广泛应用的分组密码的优化实现[8-9],对于国内分组密码算法的优化工作多见于对我国商用密码 SM 系列密码算法中的分组密码算法 SM4 的优化实现。郎欢等[10]在 2018 年提出了利用 x86 下的单指令多数据流(single instruction multiple data, SIMD)指令集选取粗粒度并行策略对 SM4 算法进行软件优化实现,优化后在 Intel Core i7 - 6700 处理器上,相较于传统的查表方法性能提高了 1.38 倍。张笑从等[11]在 2020 年提出利用比特切片方法结合 SIMD 的高级矢量扩展 2(advanced vector extension 2, AVX2)指令集对 SM4 算法进行软件优化,相较于已发表的查表优化方法,其性能提高了 1.8 倍。

分组密码软件实现方法目前主要有 3 种[12],即查表实现、切片实现和内部指令实现。查表实现是一种增大存储空间,通过将固定的计算结果进行存储,从而用寻址代替计算的一种很简单的实现技巧,同时容易受到时间攻击和 cache 攻击等侧信道攻击威胁。于 1997 年提出的比特切片技术最早用于提升数据加密标准(data encryption standard, DES)算法的软件实现性能[13],之后受到广泛重视并被用于很多分组密码算法的设计之中。比特切片技术的原理是通过模拟硬件的实现方式来进行软件实现,从而对算法本身的架构进行优化。内部指令实现方法多见于指令集优化。针对不同的中央处理器(central processing unit, CPU)和不同的算法,指令集优化的方式并不唯一。对常见的 x86 系列 CPU 而言,目前常用的指令集有 SIMD 技术的数据流单指令序列扩展(streaming SIMD extensions, SSE)/AVX 指令集等。该项技术可用于在同一操作中并行处理多组数据,分为细粒度并行和粗粒度并行两种,在分组密码特定结构与工作模式下可显著提升运算效率。此外,研究者们将比特切片技术与 SIMD 技术结合对分组密码进行软件优化实现也取得了不错的效果。Rebeiro 等[14]在 2006 年提出了结合比特切片和 SSE 指令集的 AES 快速实现,其在 Intel Core 2 处理器上实现了使用 135 个时钟周期完成一次 AES 加密任务的运算速度。Grabher 等[15]在 2008 年提出了一种基于比特切片技术和 SIMD 指令集的密码学指令集扩展,实现了比特切片技术和 SIMD 指令集两方优点的融合。

uBlock 算法是吴文玲等[16]提出的一族国产分组密码算法,其分组长度和密钥长度分别支持 128 bit 和 256 bit,根据不同分组长度和密钥长度共分为三种算法,记为 uBlock-128/128、uBlock-128/256 和 uBlock-256/256。uBlock 算法的整体设计实现了安全、效率和适应性的平衡,并且在差分分析、线性分析等多种密码分析下具有足够的安全冗余。同时算法适应各种软硬件平台,可以结合不同的软硬件条件进行高速、轻量化实现[17]。

在 uBlock 算法公开代码[18]中,其使用了位宽为 128 bit 的 SSE 指令集进行软件实现。同时该文献提出若使用 AVX2 指令集,将有望进一步提升 uBlock-256/256 算法的加解密速度,展现其性能优势。由于 uBlock 分组密码算法是 2019 年新提出的算法,目前国内外尚未有公开发表的关于 uBlock 算法软件快速实现方法的研究成果。提高 uBlock 国产分组密码算法的软件实现性能,有利于推动我国密码算法产品的研发和应用、促进实现密码算法产品的自主可控。

1　uBlock 算法结构

1.1　符号

本文使用的符号如表 1 所示。

表1 符号

Tab. 1 Symbols

符号	意义
X	n bit 明文
Y	n bit 密文
K	k bit 密钥
RK^i	n bit 轮密钥
$\boldsymbol{PL}_n$, $\boldsymbol{PR}_n$, $\boldsymbol{PL}_n^{-1}$, $\boldsymbol{PR}_n^{-1}$	$\frac{n}{16}$个字节的向量置换
$\boldsymbol{s}$, $\boldsymbol{s}^{-1}$	4 bit S 盒(逆)
$\boldsymbol{S}_n$, $\boldsymbol{S}_n^{-1}$	$\frac{n}{8}$个 S 盒(逆)的并置
$\boldsymbol{S}_k$	$\frac{k}{16}$个 S 盒的并置
$\boldsymbol{PK}_1$	16 个半字节的向量置换
$\boldsymbol{PK}_2$, $\boldsymbol{PK}_3$	32 个半字节的向量置换
$\oplus$	模 2 加运算 (异或运算)
$<<< b$	循环左移 b bit
$<<<_{32} b$	分块 32 bit 循环左移 b bit
$\|\|$	比特串的连接

1.2 uBlock 整体结构

uBlock 是一族分组密码算法,包括加密算法、解密算法、密钥扩展算法三个模块。其中,加密算法和解密算法包含 r 轮迭代,密钥扩展算法用于由初始密钥 K 生成 $r+1$ 个轮密钥。uBlock 算法的分组长度和密钥长度分别支持 128 bit 和 256 bit,记为 uBlock-128/128、uBlock-128/256 和 uBlock-256/256,它们的迭代轮数 r 分别为 16、24 和 24。

uBlock 算法整体结构为代换 - 置换(substitution-permutation,SP)结构的一种细化结构——PX 结构,全称为 Pshufb-Xor,指代向量置换和异或运算指令,其整体结构如图 1 所示。其中,S 盒为 4 bit 的非线性变换层,提供混淆作用,P 表示转换过程。PX 结构中选择分块 32 的循环移位,利用 Feistel 结构构造 16×16 的二元域最优扩散层[19]。

1.3 加密算法

加密算法由 r 轮迭代变换组成,加密轮变换如图 2 所示。

在具体操作中,输入 n bit 明文 X 和轮密钥 RK^0, RK^1, …, RK^r,输出 n bit 密文 Y。首先将明文 X 分为 X_0、X_1 两部分,而后进入轮函数。在轮函数内,轮密钥 RK^i 首先被分为 RK_0^i、RK_1^i 两部分,而后 X_0、X_1 的值更新为其分别与轮密钥的两部分异或后再经过 S 盒运算的值。之后,X_1 的值更新为 $X_0 \oplus X_1$。此后 X_0、X_1 的值需各经过两次更新,其值变为另一部分的循环左移 4 bit、8 bit、8 bit、20 bit 的值与自身值的异或结果,顺序交错执行。完成后,X_0 的值更新为 $X_0 \oplus X_1$。最后,X_0、X_1 的值分别更新为 $PL_n(X_0)$ 和 $PR_n(X_1)$。将上述轮函数过程循环 r 次,RK^i 中 i 的值取遍 0 到 $r-1$ 中的所有整数,并将 RK^r 与 X_0、X_1 拼接后的字串的异或结果输出。其形式化伪代码如算法 1 所示。

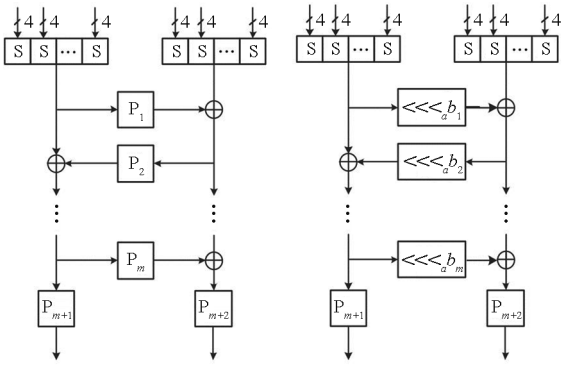

图1 PX 整体结构

Fig. 1 PX overall structure

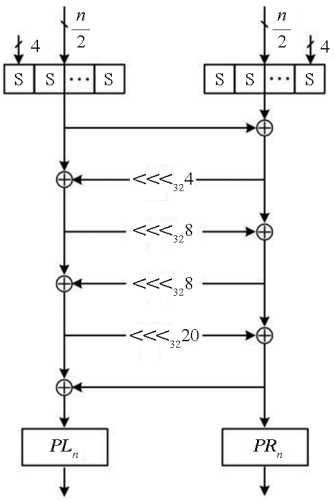

图2 加密轮变换

Fig. 2 Encrypted round transformation

其中基本模块定义如下:

1) $\boldsymbol{S}_n$:$\boldsymbol{S}_n$ 由 $n/8$ 个相同的 4 bit S 盒并置而成,定义如下:

$$\boldsymbol{S}_n:(\{0,1\}^4)^{n/8}\rightarrow(\{0,1\}^4)^{n/8}$$
$$(x_0, x_1, \cdots, x_{(n/8)-1})\rightarrow(s(x_0), s(x_1), \cdots, s(x_{(n/8)-1})) \quad (1)$$

4 bit S 盒如表 2 所示。

算法 1　加密算法

Alg. 1　Encryption algorithm

输入:n bit 明文 X,轮密钥 RK^0, RK^1, …, RK^r
输出:n bit 密文 Y

1. $X_0 \| X_1 \leftarrow X$
2. **for** $i = 0$ to $r-1$ **do**
3. $RK_0^i \| RK_1^i \leftarrow RK^i$
4. $X_0 \leftarrow S_n(X_0 \oplus RK_0^i)$
5. $X_1 \leftarrow S_n(X_1 \oplus RK_1^i)$
6. $X_1 \leftarrow X_1 \oplus X_0$
7. $X_0 \leftarrow X_0 \oplus (X_1 <<<_{32} 4)$
8. $X_1 \leftarrow X_1 \oplus (X_0 <<<_{32} 8)$
9. $X_0 \leftarrow X_0 \oplus (X_1 <<<_{32} 8)$
10. $X_1 \leftarrow X_1 \oplus (X_0 <<<_{32} 20)$
11. $X_0 \leftarrow X_0 \oplus X_1$
12. $X_0 \leftarrow PL_n(X_0)$
13. $X_1 \leftarrow PR_n(X_1)$
14. **end for**
15. $Y \leftarrow RK^r \oplus (X_0 \| X_1)$

表 2　4 bit S 盒($\boldsymbol{s}$)

Tab. 2　4 bit S-box($\boldsymbol{s}$)

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

2)$\boldsymbol{PL}_n$ 和 $\boldsymbol{PR}_n$:$\boldsymbol{PL}_n$ 和 $\boldsymbol{PR}_n$ 都是 $m = \frac{n}{16}$ 个字节的向量置换,具体见表 3。

表 3　$\boldsymbol{PL}_n$ 和 $\boldsymbol{PR}_n$

Tab. 3　$\boldsymbol{PL}_n$ and $\boldsymbol{PR}_n$

向量	取值
$\boldsymbol{PL}_{128}$	(1,3,4,6,0,2,7,5)
$\boldsymbol{PR}_{128}$	(2,7,5,0,1,6,4,3)
$\boldsymbol{PL}_{256}$	(2,7,8,13,3,6,9,12,1,4,15,10,14,11,5,0)
$\boldsymbol{PR}_{256}$	(6,11,1,12,9,4,2,15,7,0,13,10,14,3,8,5)

例如,$\boldsymbol{PL}_{128}$的表达式为:

$$\boldsymbol{PL}_{128}:(\{0,1\}^8)^8 \rightarrow (\{0,1\}^8)^8$$
$$(y_0,y_1,y_2,\cdots,y_6,y_7) \rightarrow (z_0,z_1,z_2,\cdots,z_6,z_7)$$
$$z_0 = y_1,\ z_1 = y_3,\ z_2 = y_4,\ z_3 = y_6,$$
$$z_4 = y_0,\ z_5 = y_2,\ z_6 = y_7,\ z_7 = y_5 \tag{2}$$

1.4　解密算法

解密算法由 r 轮迭代变换组成,输入n bit密文 Y 和轮密钥 RK^r, RK^{r-1}, …, RK^0,输出n bit明文 X。解密算法具体步骤与算法 1 类似,其形式化伪代码如算法 2 所示,其中,$\boldsymbol{S}_n^{-1}$、$\boldsymbol{PL}_n^{-1}$ 和 $\boldsymbol{PR}_n^{-1}$ 分别是 $\boldsymbol{S}_n$、$\boldsymbol{PL}_n$ 和 $\boldsymbol{PR}_n$ 的逆,S 盒与向量置换过程的逆运算具体见表 4 和表 5。

算法 2　解密算法

Alg. 2　Decryption algorithm

输入:n bit 密文 Y,倒序轮密钥 RK^r, RK^{r-1}, …, RK^0
输出:n bit 明文 X

1. $Y_0 \| Y_1 \leftarrow Y$
2. **for** $i = r$ to 1 **do**
3. $RK_0^i \| RK_1^i \leftarrow RK^i$
4. $Y_0 \leftarrow Y_0 \oplus RK_0^i$
5. $Y_1 \leftarrow Y_1 \oplus RK_1^i$
6. $Y_0 \leftarrow PL_n^{-1}(Y_0)$
7. $Y_1 \leftarrow PR_n^{-1}(Y_1)$
8. $Y_0 \leftarrow Y_0 \oplus Y_1$
9. $Y_1 \leftarrow Y_1 \oplus (Y_0 <<<_{32} 20)$
10. $Y_0 \leftarrow Y_0 \oplus (Y_1 <<<_{32} 8)$
11. $Y_1 \leftarrow Y_1 \oplus (Y_0 <<<_{32} 8)$
12. $Y_0 \leftarrow Y_0 \oplus (Y_1 <<<_{32} 4)$
13. $Y_1 \leftarrow Y_1 \oplus Y_0$
14. $Y_0 \leftarrow S_n^{-1}(Y_0)$
15. $Y_1 \leftarrow S_n^{-1}(Y_1)$
16. **end for**
17. $X \leftarrow RK^0 \oplus (Y_0 \| Y_1)$

表 4　4 bit S 盒的逆($\boldsymbol{s}^{-1}$)

Tab. 4　Inverse of 4 bit S-box($\boldsymbol{s}^{-1}$)

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

表 5　$\boldsymbol{PL}_n^{-1}$ 和 $\boldsymbol{PR}_n^{-1}$

Tab. 5　$\boldsymbol{PL}_n^{-1}$ and $\boldsymbol{PR}_n^{-1}$

向量	取值
$\boldsymbol{PL}_{128}^{-1}$	(4,0,5,1,2,7,3,6)
$\boldsymbol{PR}_{128}^{-1}$	(3,4,0,7,6,2,5,1)
$\boldsymbol{PL}_{256}^{-1}$	(15,8,0,4,9,14,5,1,2,6,11,13,7,3,12,10)
$\boldsymbol{PR}_{256}^{-1}$	(9,2,6,13,5,15,0,8,14,4,11,1,3,10,12,7)

1.5 密钥扩展算法

将 k bit 密钥 K 放置在 k bit 寄存器，取寄存器的左 n bit 作为轮密钥 RK^0，然后对 $i = 1, 2, \cdots, r$ 更新寄存器，并取寄存器的左 n bit 作为轮密钥 RK^i。首先将密钥 K 分为 K_0、K_1、K_2、K_3 四个部分，然后将 $K_0 \| K_1$ 经过 $\boldsymbol{PK}_t$ 运算并更新其值。之后将 K_0 与轮常数 RC_i 进行异或后经过 S 盒运算，再与 K_2 进行异或，得到的结果用于更新 K_2。接下来，再将 K_1 进行 T_k 运算，得到的结果与 K_3 进行异或运算，结果用于更新 K_3。最后将四部分进行连接，输出密钥 K。密钥扩展寄存器更新伪代码如算法 3 所示。

算法 3 密钥扩展寄存器更新

Alg. 3 Key expansion register update

输入：k bit 密钥 K

输出：k bit 密钥 K

1. $K_0 \| K_1 \| K_2 \| K_3 \leftarrow K$
2. $K_0 \| K_1 \leftarrow PK_t(K_0 \| K_1)$
3. $K_2 \leftarrow K_2 \oplus S_k(K_0 \oplus RC_i)$
4. $K_3 \leftarrow K_3 \oplus T_k(K_1)$
5. $K \leftarrow K_2 \| K_3 \| K_1 \| K_0$

其中，$\boldsymbol{S}_k$ 是 $k/16$ 个 4 bit S 盒的并置，$\boldsymbol{T}_k$ 是对 K_1 的每半字节$\otimes 2$，有限域 $GF(2^4)$ 的不可约多项式 $m(x) = x^4 + x + 1$；RC_i 为 32 bit 轮常数，作用在 K_0 的左 32 bit。$\boldsymbol{PK}_t$ 在 t 取 1，2，3 时有三种情况：$\boldsymbol{PK}_1$、$\boldsymbol{PK}_2$ 和 $\boldsymbol{PK}_3$，其分别用于 uBlock-128/128、uBlock-128/256 和 uBlock-256/256 的密钥扩展算法。$\boldsymbol{PK}_1$ 是 16 个半字节的向量置换，$\boldsymbol{PK}_2$ 和 $\boldsymbol{PK}_3$ 都是 32 个半字节的向量置换，具体向量值见表 6。轮常数 $RC_i(i = 1, 2, \cdots, 24)$ 的 16 进制值如表 7 所示。

表 6 向量置换 $\boldsymbol{PK}_t$

Tab. 6 Vector permutation $\boldsymbol{PK}_t$

向量	向量值
$\boldsymbol{PK}_1$	(6,0,8,13,1,15,5,10,4,9,12,2,11,3,7,14)
$\boldsymbol{PK}_2$	(10,5,15,0,2,7,8,13,14,6,4,12,1,3,11,9,24,25,26,27,28,29,30,31,16,17,18,19,20,21,22,23)
$\boldsymbol{PK}_3$	(10,5,15,0,2,7,8,13,1,14,4,12,9,11,3,6,24,25,26,27,28,29,30,31,16,17,18,19,20,21,22,23)

表 7 轮常数 RC_i

Tab. 7 Round constants RC_i

轮常数	值	轮常数	值
RC_1	988cc9dd	RC_{13}	dcc88d99
RC_2	f0e4a1b5	RC_{14}	786c293d
RC_3	21357064	RC_{15}	30246175
RC_4	8397d2c6	RC_{16}	a1b5f0e4
RC_5	c7d39682	RC_{17}	8296d3c7
RC_6	4f5b1e0a	RC_{18}	c5d19480
RC_7	5e4a0f1b	RC_{19}	4a5e1b0f
RC_8	7c682d39	RC_{20}	55410410
RC_9	392d687c	RC_{21}	6b7f3a2e
RC_{10}	b3a7e2f6	RC_{22}	17034652
RC_{11}	a7b3f6e2	RC_{23}	effbbeaa
RC_{12}	8e9adfcb	RC_{24}	1f0b4e5a

2 设计与实现方案

本文的代码实现包括 6 个版本，即 uBlock-128/128、uBlock-128/256 和 uBlock-256/256 三个 uBlock 版本和其各自的单密钥和多密钥版本。在此对单密钥和多密钥使用场景进行说明。

单密钥版本指用一个密钥对输入的四个分组进行加解密，在并行计算环境下，可以实现单线程下对四个分组的高效加解密运算；多密钥版本则支持使用多个密钥对输入的四个分组同时进行加解密。当选择的四个分组来源于不同的四个文件时，由于四组运算相互并行，输入输出之间互不干扰，此时对于每个分组来说是串行操作，可支持密码块链接方式(cipher block chaining mode, CBC)工作模式。

多密钥版本本质上是对单密钥版本的扩展，因此 uBlock-128/128、uBlock-128/256、uBlock-256/256 各自的单密钥和多密钥版本使用相同的优化方法。在本节中，只对单密钥版本的实现进行说明。

2.1 指令集替换

SIMD，即单指令多数据流，其构造目标是将所有数据向量化，易于批处理。较大的位宽和统一的向量化的操作使得 SIMD 指令集与大量数据的相同计算需求具有极高的相适性，较为标准的计算格式也使得 SIMD 指令集的相关函数计算效率更加优良，从而提升主机的数据处理速度[20]。

相较于支持 128 位数据宽度的 SSE 指令集，

AVX2 将可操作位宽扩展至其两倍,达到 256 位。虽然目前 Intel 处理器中已经存在比 AVX2 可操作位宽更大、性能更优越的指令集,如 AVX512 指令集,但考虑到 AVX512 指令集在密码算法优化应用中普及率低,基于 AVX512 指令集的相关产品少,同时目前大量密码算法优化相关文献仍选择使用 AVX2 指令集进行优化工作,并将其优化结果作为性能对比的指标[21],因此本文选用 AVX2 指令集进行优化实现。

本文对 uBlock-128/128、uBlock-128/256 和 uBlock-256/256 均采用 AVX2 指令集来替换原代码的 SSE 指令集。由于 AVX2 指令集相较于 SSE 指令集的可操作位宽提升了一倍,故对于以 128 bit分组作为输入的前两个 uBlock 版本而言,可实现单线程下一次性加解密两个分组;对于 256 bit分组,可减少加解密过程中使用的寄存器数量。大大体现了 uBlock 算法对于 AVX2 指令集的契合程度。

2.2　数据存储结构的优化

permute 系列指令起向量重排作用,其根据一个整形控制常量 imm8 规定分割后向量各部分重排的顺序。使用的函数决定对输入向量进行分割操作的粒度,最后将结果输出至目标向量 ***dst*** 内。例如在函数 _mm256_permute2x128_si256 (__m256i a,__m256i b,const int imm8)内,控制常数 imm8 中的每 2 bit 一组的四个分组分别对应两输入向量的四个 128 bit 分组,倒序进行输出结果 ***dst*** 的构造,同时可以指定任意一个 ***dst*** 中的 128 bit 分组为全零向量,达到重排输入向量的目的。

注意到在算法 1 的步骤 6 与算法 2 的步骤 8 均为输入组自身两部分的异或运算。

原代码中使用的 SSE 指令集数据位宽为 128 bit,为单线程加解密,无须进行额外 permute 操作。然而在 AVX2 系列指令集中,用于计算两个__m256i 变量异或的指令仅有 _mm256_xor_si256 (__m256i a,__m256i b),并且只能得到两输入变量的直接异或结果,无法进行自定义分组运算。

对于 128 bit 分组的版本,考虑到 AVX2 指令集操作均以 256 bit 为单位执行,并且若在多组运算并行的场景下将一个完整的明密文分组依照原次序存储至一个__m256i 数据结构中,则在处理后续并行分组内的组内异或运算时,必须使用较高延迟的 permute 系列指令来对两个变量进行两组临时重排,即在运算前进行一次重排,使其错开顺序,满足异或要求,运算完成后,再进行逆重排,恢复顺序排布,具体操作如下。

$$\begin{aligned}&(X_0,X_1),(X_0',X_1')\\&\xrightarrow{\text{permute}}(X_0,X_0'),(X_1,X_1')\\&\xrightarrow{\oplus}(X_0,X_0'),(X_0\oplus X_1,X_0'\oplus X_1')\\&\xrightarrow{\text{permute}}(X_0,X_0\oplus X_1),(X_0',X_0'\oplus X_1')\end{aligned}\tag{3}$$

此操作发生在轮函数内,高延迟指令与大量循环形成叠加效应,造成严重降速。故在整体数据结构上,从密钥生成到加解密轮函数的数据处理过程中,保持“左左 + 右右”型存储结构,即可避免为异或操作而临时进行的 permute 操作。由于从始至终顺序统一,运算的正确性不受影响。

对于密钥生成函数,替换示例如式 (4),对于加解密轮函数,替换示例如式 (5)。

$$\begin{aligned}&(Subkey_{1,l},\ Subkey_{1,r}),\ (Subkey_{2,l},\ Subkey_{2,r})\\&\rightarrow(Subkey_{1,l},\ Subkey_{2,l}),\ (Subkey_{1,r},\ Subkey_{2,r})\end{aligned}\tag{4}$$

$$\begin{cases}(X_0,X_1),(X_0',X_1')\rightarrow(X_0,X_0'),(X_1,X_1')\\(Y_0,Y_1),(Y_0',Y_1')\rightarrow(Y_0,Y_0'),(Y_1,Y_1')\end{cases}\tag{5}$$

所有中间计算完成后,利用一组 permute 操作来更正存储顺序,这样消除了在大量中间过程中使用高延迟 permute 系列指令所带来的降速。

在 uBlock-256/256 版本中,消息分组为 256 bit,使用 AVX2 指令集时会出现组间元素相互运算的情况,故本优化方法仅适用于 uBlock-128/128 和 uBlock-128/256 版本。

2.3　低延迟指令构造与等价替换

本文对三个算法版本的代码均进行了部分低延迟指令等价替换。下面以 uBlock-128/128 为例具体说明替换原理。

在原代码中,8 bit 分组内高 4 位为 0,以便直接进行半字节 S 盒替换。因此需先将读入的 128 bit分组通过增加高位 0 的方式“稀释”到两个__m128i 中。经过读入、移位以及通过异或初始工作向量 ***con*** 的方式添加高位 0 后,在原代码中使用了四个排序向量 ***c***1 ~ ***c***4 来进行顺序调整。

优化工作是在数据存储结构改变的基础上使用低延迟指令进行替换,两版本代码片段如表 8 所示。

优化代码片段中的_mm256_unpacklo_epi8 (__m256i a,__m256i b)和_mm256_ unpackhi_epi8 (__m256i a,__m256i b) 函数是针对两个输

入向量 ***a*** 和 ***b*** 的组内重排函数。unpacklo 函数是将两变量 lane 左右两侧 128 bit 部分的两个低 64 位 bit 进行组合输出，而 unpackhi 函数是将两变量 lane 左右两侧 128 bit 部分的两个高 64 位 bit 进行组合输出。通过查找 Intel® Intrinsics Guide[22]，计算得到如上两代码片段的总延迟如表 9 所示。

表 8 低延迟指令等价替换前后代码片段

Tab. 8 Low latency instruction equivalent substitution before and after code snippet

原代码片段	优化代码片段
t1 = _mm_shuffle_epi8 (state1, c1);	
t2 = _mm_shuffle_epi8 (state1, c2);	
t3 = _mm_shuffle_epi8 (state1, c3);	t1 = _mm256_unpacklo_epi8 (state1, state2);
t4 = _mm_shuffle_epi8 (state1, c4);	t2 = _mm256_unpackhi_epi8 (state1, state2);
state1 = _mm_xor_si128 (t1, t2);	
state2 = _mm_xor_si128 (t3, t4);	

表 9 低延迟指令等价替换前后代码延迟

Tab. 9 Low latency instruction equivalent substitution before and after code delay

处理器架构	延迟（优化前/后）	吞吐量（优化前/后）
Skylake	6/2	4.66/1
Broadwell	6/2	4.66/2
Haswell	6/2	4.66/2
Ivy Bridge	6/2	2.66/2

可见优化后，该部分总延迟较优化前降低 2/3。

2.4 S 盒的优化实现

uBlock 算法中的 S 盒查表操作是由 $n/8$ 个相同的 4 bit S 盒并置完成（其中 n 为 128 bit 或 256 bit的分组长度），故 S 盒的查表操作可在寄存器允许位宽的基础上使用并行进行优化。基于此思路，本文对三个版本代码的 S 盒查表操作均进行了优化。

利用 AVX2 指令集的 256 bit 数据位宽，可将 128 bit 分组使用一个寄存器进行存储，而256 bit 分组则需分为两部分分别存储。通过将 S 盒使用一个寄存器单独存储，分组的每部分都可进行并行 S 盒替换，最后将结果输出至目标向量 ***dst***。具体使用的指令为 AVX2 的向量重排指令_mm256_shuffle_epi8(__m256i a, __m256i b)。在该函数中，当第一个输入的参数为 S 盒，第二个参数为输入向量时，输出即为依据 S 盒得到的输出结果。这样只需对分组的左右两部分各使用一条指令即完成一轮的 S 盒查表工作。

```
state1 = _mm256_shuffle_epi8(S, state1);
state2 = _mm256_shuffle_epi8(S, state2);
```

2.5 扩散层的优化实现

2.5.1 向量置换部分

uBlock 算法中向量置换 P 是将输入的256 bit 向量以字节为单位分组，然后将分组之后的数据位置进行重新排列。

对于__m256i 数据，将左右 128 bit 之间的间隔称为 lane。在处理 128 bit 分组时，P 置换并不会涉及跨 lane 的操作。而处理256 bit分组时，分组左半部分将存放在同一个寄存器的两个 lane 中，右半部分同理。故此时，P 置换涉及跨越 lane 的操作。由于字节的数据位宽是4 bit的倍数，故类似于 S 盒置换的操作，同样可以使用_mm256_shuffle_epi8 (__m256i a, __m256i b) 指令对 uBlock-256/256 进行优化。但该指令将两个 lane 分开进行处理，不支持跨越 lane 的操作，因此相较 128 bit 向量的处理有些不同。

具体地，对于 $\boldsymbol{PL}_{256}$ 构造出如下的控制掩码。

$$\begin{aligned}\boldsymbol{pl}_0 = (&4, 5, 14, 15, 16, 17, 26, 27, 6, 7, 12, \\ &13, 18, 19, 24, 25, 2, 3, 8, 9, 30, 31, \\ &20, 21, 28, 29, 22, 23, 10, 11, 0, 1)\end{aligned} \tag{6}$$

之后将数据分为四类：

1）原位于低 128 bit，置换至低 128 bit；

2）原位于低 128 bit，置换至高 128 bit；

3）原位于高 128 bit，置换至低 128 bit；

4）原位于高 128 bit，置换至高 128 bit。

其中，属于第 1、4 类的元素不需要跨越 lane，后续运算将其放在一起处理，属于第 2、3 类的元素需要跨越 lane，也将其放在一起处理。为了分辨出属于四种不同情况的元素，可利用重排函数_mm256_shuffle_epi8 (__m256i a, __m256i b) 的特点，构造如下选择向量进行不同类型元素的处理。

$$\boldsymbol{K}_0 = (\underbrace{0\text{x}70, \cdots, 0\text{x}70}_{16}, \underbrace{0\text{x}F0, \cdots, 0\text{x}F0}_{16}) \tag{7}$$

$$\boldsymbol{K}_1 = (\underbrace{0\text{xF0},\cdots,0\text{xF0}}_{16},\underbrace{0\text{x70},\cdots,0\text{x70}}_{16}) \quad (8)$$

其中,$\boldsymbol{K}_0$ 选择出不需要跨越 lane 的第 1、4 类,$\boldsymbol{K}_1$ 选择出需要跨越 lane 的第 2、3 类。对于第 2、3 类,需要将置换之前的向量先交换高低128 bit,然后再进行置换操作。将控制数置为 0x4E 可以实现高低 128 bit 的交换。

整体的 256 bit 向量置换伪代码如算法 4 所示。

算法 4　256 bit 向量置换

Alg. 4　256 bit vector permutation

输入:待置换向量__ m256i value,控制向量__ m256i shuffle

输出:leftPart, rightPart

```
1. DEFINE Shuffle (__m256i value, __shuffle)
2.     add0 = = _mm256_add_epi8(shuffle, K0);
3.     add1 = = _mm256_add_epi8(shuffle, K1);
4.     shu0 = = _mm256_shuffle_epi8(value, add0);
5.     per1 = = _mm256_permute4x64_epi64(value, 0x4E);
6.     shu1 = = _mm256_shuffle_epi8(per1, add1);
7.     return_mm256_xor_si256(shu0, shu1);
8. L1 = (4,5,14,15,16,17,26,27,6,7,12,13,18,19,24,
   25, 2,3,8,9,30,31,20,21,28,29,22,23,10,11,0,1);
9. L2 = (12,13,22,23,2,3,24,25,18,19,8,9,4,5,30,
   31,14, 15,0,1,26,27,20,21,28,29,6,7,16,17,10,
   11);
10. leftPart = Shuffle(leftPart, L1);
11. rightPart = Shuffle(rightPart, L2);
```

注意到其中加法的部分每次运算都是一样的,可以提前计算出来,故简化算法如算法 5 所示。

2.5.2　循环移位部分

三个版本代码中移位位数分别为 4 位、8 位、20 位,均为 4 的倍数。若按 4 位分组,则输入的每一组在循环移位后会对应到相应的输出组。故循环移位可通过_mm256_shuffle_epi8(__m256i a, __m256i b)实现,其第一个参数为循环移位前的分组,掩模可设置为元素对应输入前的位置,例如使一个元素循环左移四位的掩模 A1 如下,每轮四次的循环移位操作对应四个向量的四次重排。

```
__m256i A1 = _mm256_setr_epi8(1, 2, 3, 4, 5,
                6,7, 0, 9, 10, 11, 12, 13, 14, 15,
                8, 17, 18, 19, 20, 21, 22, 23, 16,
                25, 26, 27, 28, 29, 30, 31, 24)
```

(9)

算法 5　简化后的 256 bit 向量置换

Alg. 5　Simplified 256 bit vector permutation

输入:待置换向量__ m256i state1,__ m256i state2

输出:置换后向量 state1,state2

```
1. L1_front = (0x74,0x75,0x7e,0x7f,0x80,0x81,0x8a,
   0x8b,0x76,0x77,0x7c,0x7d,0x82,0x83,0x88,0x89,
   0xf2,0xf3,0xf8,0xf9,0x0e,0x0f,0x04,0x05,0x0c,
   0x0d,0x06,0x07,0xfa,0xfb,0xf0,0xf1);
2. L1_rear = (0xf4,0xf5,0xfe,0xff,0x00,0x01,0x0a,
   0x0b,0xf6,0xf7,0xfc,0xfd,0x02,0x03,0x08,0x09,
   0x72, 0x73,0x78,0x79,0x8e,0x8f,0x84,0x85,0x8c,
   0x8d, 0x86,0x87,0x7a,0x7b,0x70,0x71);
3. L2_front = (0x7c,0x7d,0x86,0x87,0x72,0x73,0x88,
   0x89,0x82,0x83,0x78,0x79,0x74,0x75,0x8e,0x8f,
   0xfe,0xff,0xf0,0xf1,0x0a,0x0b,0x04,0x05,0x0c,
   0x0d,0xf6,0xf7,0x00,0x01,0xfa,0xfb);
4. L2_rear = (0xfc,0xfd,0x06,0x07,0xf2,0xf3,0x08,
   0x09,0x02,0x03,0xf8,0xf9,0xf4,0xf5,0x0e,0x0f,
   0x7e,0x7f,0x70,0x71,0x8a,0x8b,0x84,0x85,0x8c,
   0x8d,0x76, 0x77,0x80,0x81,0x7a,0x7b);
5. t1 = _mm256_shuffle_epi8(state1, L1_front);
6. t3 = _mm256_permute4x64_epi64(state1, 0x4E);
7. t2 = mm256 shuffle epi8(t3, L1_rear);
8. state1 = _mm256_xor_si256(t1, t2);
9. t1 = _mm256_shuffle_epi8(state2, L2_front);
10. t3 = _mm256_permute4x64_epi64(state2, 0x4E);
11. t2 = _mm256_shuffl_epi8(t3, L2_rear);
12. state2 = _mm256_xor_si256(t1, t2);
```

2.5.3　异或部分

使用指令_mm256_xor_si256(__m256i a, __m256i b)快速完成 32 B 异或,并输出结果。

2.6　并行运算实现

uBlock 算法中的所有操作均为半字节操作,故原代码中寄存器每 8 位只有低 4 位有值,高 4 位均为空值 0 而未被利用。在使用数据位宽加倍的 AVX2 指令集后,可继续利用高 4 位存放下一轮待处理的数据,以实现单线程下密钥的并行扩展及分组的并行加解密。三个版本代码均是如此。

2.6.1　并行密钥扩展

在进行密钥扩展时,在高位中放入下一个待扩展的密钥,以达到同时扩展的目的。需要指出的是,高 4 位的使用对 xor 函数、and 函数以及普通的 shuffle 函数等操作均不会有影响。但由于 uBlock 算法半字节操作的特性,S 盒操作和紧接着的 T_k 操作均会受到影响。因此在每轮变

换中，只在 S 盒操作前将寄存器的当前值又重新分为由高位和低位组成的新的两组半字节数组。在 T_k 操作之后，再将高、低位数组合并，进行后续的其他运算。在每轮变换中，输入数据只需分开一次即可。另外，由于高位也需要参与到所有的运算中，代码中的常量 RC_i 也需要有所改变，其高位应设置为与低位相同的值，例如：0x09 → 0x99。

2.6.2 并行加解密

在进行加解密时，在高位中放入后一个或两个分组。对于 uBlock-128/128 版本，优化后的代码可以同时加解密四个分组；对于 uBlock-256/256 版本，则可以同时加解密两个分组。而每轮所进行的操作只是少量增加，这样会使加解密性能得到极大提升。需要指出的是，由于涉及密钥的操作只有明密文和密钥的异或，因此只需要保证将明密文和其所对应的密钥放在相对应的位置即可。在进行 S 盒运算时依然需要经过先分开、分别查表、最后合并的过程，而其他运算不受影响。

经上述优化后，原先未被利用的高位 0 现可被利用，在 uBlock-128/128、uBlock-128/256 中可实现单线程四个分组的加解密运算，在 uBlock-256/256 中可实现单线程两个分组的加解密运算。

2.7 编译优化

2.7.1 针对编译器的优化方案

编译时，编译参数设置为：-Ofast-mfma-mavx2-funroll-loops-march = native，其可以使编译器在编译时根据 CPU 型号等情况进行较为完整的优化，提升整体运行速度。

2.7.2 针对函数调用的优化方案

在函数前使用 inline 内联方式进行加速。在基础实现中，循环内多次调用子函数，存在函数大量调用的问题。故可在函数前加上 inline 字段，使其成为内联函数，解决频繁调用导致大量消耗栈空间的问题，使程序速度得到明显提升[23-24]。

3 实验及结果

3.1 实验环境和测试数据的选择

测试环境：Microsoft Windows 10 Professional Edition 20H2 Build 19042.1466。

CPU：AMD Ryzen 9 5900X @3.70 GHz。

程序运行内存：32 GB。

编译器：gcc（x86_64-win32-seh-rev0，Built by MinGW-W64 project）8.1.0。

系统测试方案：分别对 uBlock-128/128、uBlock-128/256 和 uBlock-256/256 进行测试。每个版本设置了在单/多密钥、短/长消息情景下的加/解密测试，故每个版本测试 8 轮（例如单密钥短消息加密测试为 1 轮），为保证测试结果的有效性，每轮测试进行 10 次后取平均数值。

测试使用数据量：短消息分组为 128 bit 或 256 bit，循环加密 1 000 000 轮为一次测试。长消息分组为 2 Mbit 数据，循环加密 1 024 轮为一次测试，在普通加解密模式和 CBC 模式中使用。

3.2 测试结果

按照上述方案进行测试后，uBlock-128/128、uBlock-128/256 和 uBlock-256/256 优化前后速度对比如图 3、图 4、图 5 所示。其中，AVX2 平凡实现测试的代码只在原代码基础上进行了 AVX2 指令集替换，并未进行其他优化，以此来比较除指令集替换之外其他优化方法的效果。

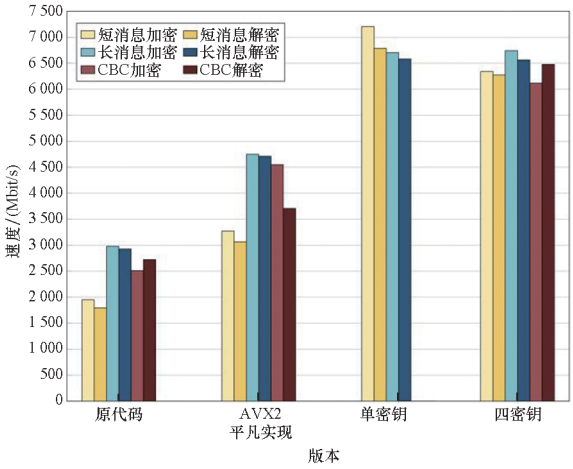

图 3 uBlock-128/128 速度对比图

Fig. 3 uBlock-128/128 speed comparison figure

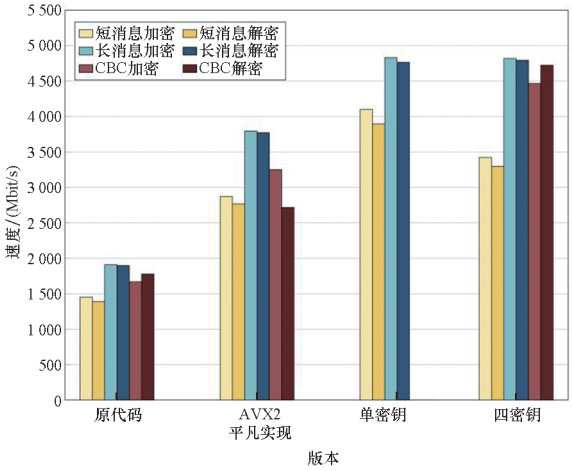

图 4 uBlock-128/256 速度对比图

Fig. 4 uBlock-128/256 speed comparison figure

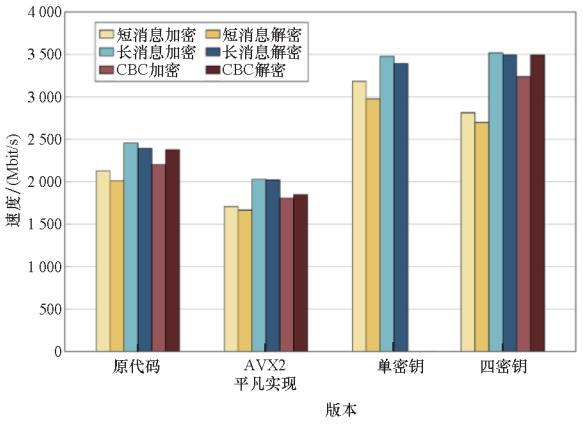

图 5　uBlock-256/256 速度对比图

Fig. 5　uBlock-256/256 speed comparison figure

分析图表,可以发现:

1)在 CBC 四密钥加密模式下,uBlock-128/128、uBlock-128/256 和 uBlock-256/256 三个算法版本运行速度较原代码分别提升 143%、167% 和 47%,较 AVX2 平凡实现分别提升 34%、37% 和 79%。

2)图 3 显示,在 uBlock-128/128 单密钥版本中,长消息加解密速度慢于短消息。针对这样的反常情况,进行了在不同电脑环境下的反复测试。结果表明,性能较好的电脑普遍更倾向于反常情况。这可能是由于在进行短消息加解密时,每轮使用的数据相同,存储位置未变,因此磁头无须移动;而进行长消息加解密时,每轮需要依次读取多个分组,每个分组存储位置不同,因此磁头需要持续移动。由于 uBlock-128/128 处理的数据量与另外两个版本相比而言更少,因此磁头的移动对速度的影响成为主导因素。而对于 uBlock-128/256 和 uBlock-256/256,由于密钥扩展处理的数据量更大,因此密钥扩展对速度的影响成为主导因素,故长消息加解密速度明显快于短消息。

3)在 uBlock-256/256 中,AVX2 平凡实现的速度较原代码慢。这是因为在 AVX2 平凡实现中未使用 2.2 节中改变输入顺序的优化方法,需要使用较多的 permute 系列指令来保证程序正确性,同时在未扩展高位的情况下无法实现两组并行。综合两者原因导致了 uBlock-256/256 AVX2 平凡实现的速度较慢。

4　结论

本文通过使用支持 256 bit 数据位宽的 AVX2 指令集,提高编译器自动优化等级,优化数据存储结构,综合使用高位并行、低延迟指令逻辑优化等方法实现了单线程并行计算,对国产分组密码 uBlock 算法的软件性能进行了优化。

在 AMD Ryzen 9 5900X @ 3.70 GHz 环境下进行性能测试,结果显示:

1)uBlock-128/128 算法单密钥短消息加密速度达到 7 205 Mbit/s,较原代码提升 269%;四密钥短消息加密速度达到 6 340 Mbit/s,较原代码提升 225%。

2)uBlock-128/256 算法单密钥短消息加密速度达到 4 099 Mbit/s,较原代码提升 182%;四密钥短消息加密速度达到 3 422 Mbit/s,较原代码提升 136%。

3)uBlock-256/256 算法单密钥短消息加密速度达到 3 182 Mbit/s,较原代码提升 49%;四密钥短消息加密速度达到 2 813 Mbit/s,较原代码提升 32%。

优化后的算法在未来的工程实现中将具有广泛的硬件基础,同时作为国产密码满足了自主可控的需求。最后,对于三个版本的代码,均实现了单密钥与多密钥版本,适应不同的加解密场景。

参考文献(References)

[1] 王姗姗, 徐雷, 张曼君. 浅析网络安全发展趋势[J]. 通信世界, 2022(2): 22 – 24.
WANG S S, XU L, ZHANG M J. Analysis on the development trend of network security[J]. Communications World, 2022(2): 22 – 24. (in Chinese)

[2] QADIR A M, VAROL N. A review paper on cryptography[C]// Proceedings of the 7th International Symposium on Digital Forensics and Security (ISDFS), 2019.

[3] SAINI N, MANDAL S. Review paper on cryptography[J]. International Journal of Research, 2015, 2(5): 45 – 49.

[4] RANA M, MAMUN Q, ISLAM R. Lightweight cryptography in IoT networks: a survey [J]. Future Generation Computer Systems, 2022, 129: 77 – 89.

[5] 仲利波, 赵婧琳. 美国密码技术的出口管制法律制度研究[J]. 信息安全研究, 2018, 4(3): 211 – 218.
ZHONG L B, ZHAO J L. The evolution and enlightenment of US encryption export controls legal system[J]. Journal of Information Security Research, 2018, 4(3): 211 – 218. (in Chinese)

[6] 姗姗. 使用国产密码技术是应对“棱镜”首选[J]. 计算机与网络, 2013, 39(21): 6 – 7.
SHAN S. Using domestic cryptographic technology is the first choice to deal with “prism”[J]. Computer & Network, 2013, 39(21): 6 – 7. (in Chinese)

[7] 公丽丽, 张文涛, 包珍珍, 等. 轻量级分组密码 RECTANGLE 在 X86 和 X64 平台的软件实现评估[J]. 中国科学院大学学报, 2015, 32(6): 816 – 824.
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. (in Chinese)

[8] WEI M M, YANG G Q, KONG F Y. Software implementation and comparison of ZUC-256, SNOW-V, and AES-256 on RISC-V platform[C]//Proceedings of IEEE International Conference on Information Communication and Software Engineering (ICICSE), 2021.

[9] SUO S L, CUI C, JIAN G Y, et al. Implementation of the high-speed SM4 cryptographic algorithm based on random pseudo rounds[C]//Proceedings of IEEE International Conference on Information Technology, Big Data and Artificial Intelligence (ICIBA), 2020.

[10] 郎欢, 张蕾, 吴文玲. SM4 的快速软件实现技术[J]. 中国科学院大学学报, 2018, 35(2): 180 – 187.
LANG H, ZHANG L, WU W L. Fast software implementation of SM4[J]. Journal of University of Chinese Academy of Sciences, 2018, 35(2): 180 – 187. (in Chinese)

[11] 张笑从, 郭华, 张习勇, 等. SM4 算法快速软件实现[J]. 密码学报, 2020, 7(6): 799 – 811.
ZHANG X C, GUO H, ZHANG X Y, et al. Fast software implementation of SM4[J]. Journal of Cryptologic Research, 2020, 7(6): 799 – 811. (in Chinese)

[12] DIXIT P, ZALKE J, ADMANE S. Speed optimization of AES algorithm with hardware-software co-design[C]//Proceedings of the 2nd International Conference for Convergence in Technology, 2017

[13] BIHAM E. A fast new DES implementation in software[C]//Proceedings of the 4th International Workshop on Fast Software Encryption, 1997.

[14] REBEIRO C, SELVAKUMAR D, DEVI A S L. Bitslice implementation of AES[C]//Proceedings of the 5th Cryptology and Network Security International Conference, 2006.

[15] GRABHER P, GROßSCHÄDL J, PAGE D. Light-weight instruction set extensions for bit-sliced cryptography[C]//Proceedings of the 10th International Workshop on Cryptographic Hardware and Embedded Systems, 2008.

[16] 吴文玲, 张蕾, 郑雅菲, 等. 分组密码 uBlock[J]. 密码学报, 2019, 6(6): 690 – 703.
WU W L, ZHANG L, ZHENG Y F, et al. The block cipher uBlock[J]. Journal of Cryptologic Research, 2019, 6(6): 690 – 703. (in Chinese)

[17] 吴文玲. 分组密码专刊序言(中英文)[J]. 密码学报, 2019, 6(6): 687 – 689.
WU W L. Preface of special issue on block cipher[J]. Journal of Cryptologic Research, 2019, 6(6): 687 – 689. (in Chinese)

[18] 吴文玲, 张蕾, 郑雅菲, 等. 分组密码 uBlock 详细设计[EB/OL]. (2019 – 02 – 01)[2022 – 03 – 02]. https://sfjs.cacrnet.org.cn/site/content/387.html.
WU W L, ZHANG L, ZHENG Y F, et al. Detailed design of uBlock block cipher[EB/OL]. (2019 – 02 – 01)[2022 – 03 – 02]. https://sfjs.cacrnet.org.cn/site/content/387.html. (in Chinese)

[19] GUO Z Y, WU W L, GAO S. Constructing lightweight optimal diffusion primitives with Feistel structure[C]//Proceedings of the 22nd International Conference on Selected Areas in Cryptography, 2015.

[20] AMIRI H, SHAHBAHRAMI A. SIMD programming using Intel vector extensions[J]. Journal of Parallel and Distributed Computing, 2020, 135: 83 – 100.

[21] 杨昊, 刘哲, 黄军浩, 等. AKCN-MLWE 算法 AVX2 高效实现[J]. 计算机学报, 2021, 44(12): 2560 – 2572.
YANG H, LIU Z, HUANG J H, et al. High-speed AVX2 implementation of AKCN-MLWE[J]. Chinese Journal of Computers, 2021, 44(12): 2560 – 2572. (in Chinese)

[22] Intel Corporation. Intel® intrinsics guide v3.6.7[EB/OL]. (2023 – 12 – 07)[2022 – 03 – 02]. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.

[23] ALLEN R, JOHNSON S. Compiling C for vectorization, parallelization, and inline expansion[J]. ACM SIGPLAN Notices, 1988, 23(7): 241 – 249.

[24] 杨光宇, 高晓蓉, 王黎, 等. 基于 TI C6000 系列 DSP 的 C/C++ 程序优化技术[J]. 现代电子技术, 2009, 32(8): 56 – 59.
YANG G Y, GAO X R, WANG L, et al. Technology of C/C++ program optimization based on TI C6000 DSP[J]. Modern Electronics Technique, 2009, 32(8): 56 – 59. (in Chinese)