计算机研究与发展
Journal of Computer Research and Development
DOI: 10.7544/issn1000-1239.202220531
61(8): 2097−2109, 2024

面向 SIMD 指令集的 SM4 算法比特切片优化

王 闯 丁 滟 黄辰林 宋连涛
(国防科技大学计算机学院 长沙 410073)
(clinale@foxmail.com)

Bitsliced Optimization of SM4 Algorithm with the SIMD Instruction Set

Wang Chuang, Ding Yan, Huang Chenlin, and Song Liantao
(*College of Computer Science and Technology*, *National University of Defense Technology*, *Changsha* 410073)

Abstract SM4 algorithm is a commercial block cipher algorithm independently designed by China, and its encryption and decryption performance has become one of the critical factors affecting the data confidentiality of the information system. The existing optimizations mainly focus on hardware designs and software look-up tables, which have problems such as dependence on specific hardware environments, low efficiency, and vulnerability to side-channel attacks. Bit slicing technology efficiently processes block ciphers in parallel by reorganizing input data, and can resist side-channel attacks against caches. However, the existing researches on bitsliced block ciphers are highly dependent on the hardware platforms and only support a single processor architecture, and the parallel processing pipeline starts slowly. It is difficult for the encryption and decryption operations for small-scale data to give full play to the advantages of advanced instruction sets such as SIMD (single instruction multiple data) instructions. To resolve the above problems, we firstly propose a cross-platform general bitsliced block cipher algorithm model, which supports a general data slicing method that provides consistent data slicing for different processor instructions. Based on that, a fine-grained bitsliced SM4 optimization algorithm for SIMD instructions is proposed, which can effectively shorten the startup time of the algorithm through fine-grained plaintext slicing reorganization and linear transformation optimization. The experiments show that, compared with the look-up table-based SM4 algorithm, the encryption rate can reach up to 438.0 MBps. The clock cycles required for encrypting a byte are up to 7.0 CPB (cycle/B), and the encryption performance is improved by an average of 80.4% to 430.3%.

Key words SM4 algorithm; performance optimization; bit slice; side-channel attacks; SIMD instruction set

摘 要 SM4 算法是中国自主设计的商用分组密码算法,其加解密计算性能成为影响信息系统数据机密性保障的重要因素之一.现有 SM4 算法优化主要面向硬件设计和软件查表等方向展开研究,分别存在依赖特定硬件环境、效率低下且易遭受侧信道攻击等问题.比特切片技术通过对输入数据重组实现了并行化高效分组密码处理,可以抵御针对缓存的侧信道攻击.然而现有切片分组密码研究对硬件平台相关性强、处理器架构支持单一,并且并行化处理流水启动较慢,面向小规模数据的加解密操作难以充分发挥单指令多数据(single instruction multiple data, SIMD)等先进指令集的优势.针对上述问题,首先提出了一种

收稿日期: 2022-06-11; **修回日期**: 2023-11-10
基金项目: 国家自然科学基金联合基金项目(U19A2060); 国家自然科学基金项目(62172431); 基础加强计划重点研究基础研究项目(2019-XXX-ZD-188-00); 湖南省研究生科研创新项目(CX20220056)
This work was supported by the Joint Funds of the National Natural Science Foundation of China (U19A2060), the National Natural Science Foundation of China (62172431), the National Defense Science and Technology Foundation Strengthening Project of China (2019-XXX-ZD-188-00), and the Hunan Provincial Innovation Foundation for Postgraduate (CX20220056).
通信作者: 丁滟(yanding@nudt.edu.cn)

跨平台的通用切片分组密码算法模型,支持面向不同的处理器指令字长提供一致化的通用数据切片方法.在此基础上,提出了一种面向SIMD指令集的细粒度切片并行处理SM4优化算法,通过细粒度明文切片重组与线性处理优化有效缩短算法启动时间.实验结果表明,相比通用SM4算法,优化的SM4比特切片算法加密速率最高可达438.0 MBps,加密每字节所需的时钟周期最快高达7.0 CPB(cycle/B),加密性能平均提升80.4%~430.3%.

关键词 SM4算法;性能优化;比特切片;侧信道攻击;SIMD指令集

中图法分类号 TP391

随着云计算和大数据技术的发展,数据存储与处理由云服务方完成,在数据所有权和使用权分离的场景下,如何保障数据的机密性已经成为重要问题[1].当前普遍采用分组加密算法为数据提供快速对称加密,为信息系统提供数据安全基础保障.SM4算法[2-4]作为中国自主设计的分组密码算法,于2021年6月被纳入ISO/IEC国际标准,为众多信息系统提供安全、完整的数据加密方案,具有广泛的应用前景.

为了实现较好的混淆效果和增强算法安全性,分组密码算法通常使用复杂的非线性代替变换,但是这也导致了算法的软件实现性能较低,无法满足云计算数据传输场景下的高吞吐量需求.自SM4算法被公布以来,针对SM4算法的研究不断涌现.在性能优化方面,研究者们主要从硬件架构和软件实现2方面对SM4展开研究.

在硬件架构研究中,研究者们提出了循环型和非循环型[5]以及折叠型和流水线型[6-8]等硬件实现架构,极大地提升了SM4算法的硬件加解密性能.然而,相比软件优化实现,硬件优化依赖特定的硬件环境,并且需要在软件层面进行相应的适配工作,这限制了SM4硬件优化实现的应用范围.

在软件优化研究中,研究者们重点探索了基于查找表和基于比特切片的优化实现2种方案.郎欢等人[9]将非线性变换与线性变换结合在一起,提出了2种查表优化方案,并在x86平台上使用SIMD指令高效实现了SM4算法.Kwon等人[10]则针对8 b AVR微控制器、32 b RISC-V处理器和64 b ARM处理器特性,提出了面向特定平台的查表优化方案,取得了较好的优化效果.然而,由于查表优化方案使用查找表实现SM4算法,面临着针对缓存的计时攻击等侧信道攻击的威胁,进而有可能导致密钥泄露.基于比特切片的优化方案使用逻辑指令等价实现密码算法,无需查找表,能够抵抗针对缓存的侧信道攻击.

比特切片技术由Biham[11]提出,用于提升DES算法的软件实现性能,现已被广泛应用到各种密码算法的软件优化实现中,如AES[12-14],whirlpool[15],KASUMI[16]等,用于加速密码算法的软件实现性能.在SM4算法的比特切片优化实现中,Zhang等人[17]利用塔域$GF(((2^2)^2)^2)$降低实现S盒所需的代数复杂度,并应用比特切片技术给出了SM4算法在x86平台上的64分组数据块并行加解密,取得了较好的加速效果.张笑从等人[18]将S盒等价表达为逻辑函数,通过构造新的选择函数,降低了实现S盒所需的门电路数量,并在x86平台上实现了SM4算法的256分组数据并行加解密.

通过上述研究成果可以发现,目前关于SM4比特切片算法的研究大多是针对具体平台和指令集设计切片算法,缺乏通用性,并且已有的切片分组密码算法数据并行粒度较大,当利用指令操作数更长的指令集实现切片分组密码算法时(例如SIMD指令集),需要持续等待多个明文分组的到来并对其进行切片,导致加密算法启动时间较长,在对小规模数据进行加解密操作时难以充分发挥SIMD等高级指令集的优势.

本文的主要贡献包括4个方面:

1)通过对分组密码算法和处理器指令字长进行建模,提出了一种通用的切片分组密码算法模型,对在不同指令集架构下实现的切片分组密码算法进行统一建模,能够高效完成数据块的切片操作;

2)基于通用切片分组密码算法模型,针对SIMD指令集特性,结合SM4算法结构设计了一种新颖的细粒度并行的切片SM4算法,能够对小规模的数据块提供细粒度并行,加快密码算法启动时间;

3)通过对SM4算法中的线性变换进行分析,结合切片数据结构,设计了一种新颖的接收切片数据输入的切片线性变换函数,能够高效完成切片数据的线性变换;

4)针对64 b ARM平台,利用NEON指令集实现了SM4比特切片算法原型.实验结果表明,相比普通查表SM4算法,SM4比特切片算法的加密性能平均

提升了 80.4%~430.3%, 加密速率最高可达 438.0 MBps, 加密每字节所需的时钟周期最快高达 7.0 CPB (cycle/B), 适用于对密码性能要求较高的设备.

1 相关技术

在本节中, 我们主要介绍 SM4 加密算法、比特切片技术和 SIMD 指令集.

1.1 SM4 加密算法

作为一个分组加密算法, SM4 算法的分组长度和初始密钥长度均为 128 b. 加密算法和密钥扩展算法均采用 32 轮迭代结构, 解密算法和加密算法结构相同, 只需将加密轮密钥逆序输入参与迭代运算即可得到明文. SM4 加密算法过程如图 1 所示.

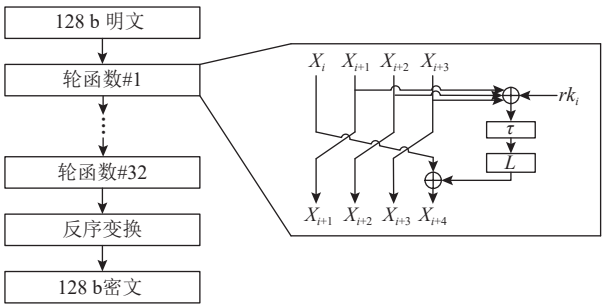


Fig. 1 Encryption algorithm flow of SM4

图 1 SM4 加密算法流程

设 SM4 明文输入为 $\boldsymbol{X}=(X_0,X_1,X_2,X_3)\in(\mathbb{Z}_2^{32})^4$, 密文输出为 $\boldsymbol{Y}=(Y_0,Y_1,Y_2,Y_3)\in(\mathbb{Z}_2^{32})^4$, 轮密钥为 $rk_i\in\mathbb{Z}_2^{32}$, $i=0,1,\cdots,31$, 则 SM4 加密算法表示为

$$\begin{aligned}&X_{i+4}=F(X_i,X_{i+1},X_{i+2},X_{i+3},rk_i)=\\&\quad X_i\oplus T(X_{i+1}\oplus X_{i+2}\oplus X_{i+3}\oplus rk_i),\\&\boldsymbol{Y}=(Y_0,Y_1,Y_2,Y_3)=(X_{35},X_{34},X_{33},X_{32}),\end{aligned}\tag{1}$$

其中 T 是一个 $\mathbb{Z}_2^{32}\to\mathbb{Z}_2^{32}$ 的合成变换, 包含非线性变换 L 和非线性变换 τ, 即 $T(\boldsymbol{x})=L(\tau(\boldsymbol{x}))$.

非线性变换 τ 是 4 个并行的 8 b 进 8 b 出的 S 盒置换. 设输入为 $\boldsymbol{x}=(x_0,x_1,x_2,x_3)\in(\mathbb{Z}_2^8)^4$, 输出为 $\boldsymbol{y}=(y_0,y_1,y_2,y_3)\in(\mathbb{Z}_2^8)^4$, 则

$$\boldsymbol{y}=(y_0,y_1,y_2,y_3)=\tau(\boldsymbol{x})=(S(x_0),S(x_1),S(x_2),S(x_3)).$$

线性变换 L 接收非线性变换 τ 的输出作为输入, 设线性变换 L 输出为 $z\in\mathbb{Z}_2^{32}$, 则

$$z=L(\boldsymbol{y})=\boldsymbol{y}\oplus(\boldsymbol{y}\lll 2)\oplus(\boldsymbol{y}\lll 10)\oplus(\boldsymbol{y}\lll 18)\oplus(\boldsymbol{y}\lll 24),\tag{2}$$

其中 $\lll i$ 表示 32 b 字循环左移 i 位, 例如 $\boldsymbol{y}\lll 2$ 表示将 32 b $\boldsymbol{y}$ 循环左移 2 b.

1.2 比特切片技术

比特切片技术是一种在软件层面上通过对输入数据重组实现了并行化高效分组密码算法处理的有效技术, 可以抵御针对缓存的侧信道攻击. 该技术的主要思想是在寄存器中存入位于多个明文分组中相同位置的比特数据, 通过操作寄存器实现多个明文分组比特数据的并行处理. 当在 N 位处理器上实现比特切片算法时, 与将明文分组放入不同寄存器中以串行方式处理的传统方法不同, 在比特切片实现中, 寄存器中的每个位都充当 1 b 处理器, 执行不同明文分组的加密, 因此可以并行加密 N 个明文分组, 为密码算法提供了并行处理能力.

与传统的基于查表的密码算法实现相比, 基于比特切片的算法实现仅包含逻辑操作, 不涉及在内存中缓存查找表, 因此可以抵抗针对缓存的侧信道攻击. 此外, 由于比特切片技术实现了同一明文分组内部不同位置的比特分离和不同明文分组内部相同位置的比特聚合, 因此在切片数据操作中可以简单地忽略某些位级操作, 例如移位操作, 这些位级操作可以通过寄存器重命名来实现相同的效果. 但是, 在切片数据表示下, 密码算法中的非线性 S 盒部分无法再作为查找表来执行, 必须利用逻辑指令等价实现 S 盒. 此外, 从常规存储结构到切片存储结构的数据转换会在密码算法加密前后引入额外的计算开销, 因此设计高效的数据切片算法能有效提升切片分组密码算法的性能表现.

1.3 SIMD 指令集

SIMD 指令是指同时对多个数据操作数执行相同操作的硬件指令. 通常, SIMD 指令接收 2 个向量作为输入(每个向量都有一组操作数), 对 2 组操作数(每个向量的一个操作数)执行相同的操作, 并输出带有结果的向量. 图 2 说明了一个 SIMD 指令并行执行 4 个操作的简单示例.

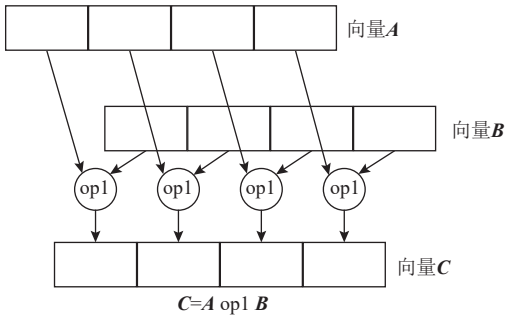


Fig. 2 Example of SIMD operation

图 2 SIMD 操作示例

自1996年以来, Intel对其x86指令集架构进行扩展, 推出了MMX、流式SIMD扩展(streaming SIMD extensions, SSE)和高级向量扩展(advanced vector extensions, AVX)指令集, 以提高多媒体应用程序和其他需要图像和信号处理的应用程序的性能. ARM处理器则通过其NEON技术在ARM-Cortex架构中引入了SIMD扩展. NEON指令集为128 b宽, 包括16或32个128 b寄存器. 这些寄存器可以被认为是相同数据类型元素的向量, 数据类型包括有符号或无符号8 b、16 b、32 b、64 b和单精度浮点.

SIMD单元中的运算通常包括基本的算术运算(如加法、减法、乘法等)、逻辑运算(如逻辑与、逻辑或、逻辑异或等)和其他运算, 如绝对值(abs)和平方根(sqrt). 表1列出了部分NEON汇编指令与SSE汇编指令, 并对这些指令进行了指令说明.

Table 1 SIMD Instruction Sets for x86 and ARM Processors

表1 x86与ARM处理器上的SIMD指令集

NEON汇编指令	SSE汇编指令	指令描述
SHL	PSLLD	并行4路32 b数据左移
USHR	PSRLD	并行4路32 b数据无符号右移
SSHR	PSRAD	并行4路32 b数据有符号右移
AND	PAND	并行4路32 b数据与运算
EOR	PXOR	并行4路32 b数据异或运算
TBL	PSHUFB	并行16路8 b数据查表

2 通用切片分组密码算法模型

本节对分组密码算法进行建模, 并提出一种通用的切片分组密码算法设计模型.

2.1 基本定义

分组密码算法是对固定长度的一组明文进行加密的算法, 其数学模型是将明文消息编码表示后的数字序列划分为长度为n的明文分组, 每组分别在密钥的控制下变换成等长的密文数字序列. 切片分组密码算法是对分组密码算法应用比特切片技术的密码算法, 其通过数据重组能并行加密多个明文分组, 因此可加速分组密码算法的实现性能. 在详细介绍如何将分组密码算法设计为切片分组密码算法前, 我们首先给出一些基本的符号解释和定义.

定义 1. 分组密码算法. 给定一个五元组(V_n, V_p, E, D, V_m), V_n代表n维明文空间, V_p代表p维密钥空间, E代表加密函数, D代表解密函数, V_m代表m维密文空间. $E(\boldsymbol{p},\boldsymbol{k})\in V_m$表示每个明文消息$\boldsymbol{p}\in V_n$在密钥$\boldsymbol{k}\in V_p$的作用下经过加密函数处理后都对应一个特定的密文. $D(\boldsymbol{c},\boldsymbol{k})\in V_n$表示每个密文消息$\boldsymbol{c}\in V_m$在密钥$\boldsymbol{k}\in V_p$的作用下经过解密函数处理后都对应一个特定的明文. 当明文分组长度n大于密文分组长度m时, 称其为有数据压缩的分组密码; 若$n<m$, 则称其为有数据扩展的分组密码. 通常情况下, 分组密码算法的明文分组长度n等于密文分组长度m.

例如, 在SM4算法中, 其明文分组长度、密文分组长度和密钥长度均为128 b; 而在AES算法中, 明文分组长度和密文分组长度均为128 b, 密钥长度包含可选的128 b, 192 b, 256 b.

定义 2. 分组密码算法中的l阶数据块. 我们将分组密码算法中的l阶数据块表示为$\boldsymbol{B}_{1\times l}=(\boldsymbol{B}_1,\boldsymbol{B}_2,\cdots,\boldsymbol{B}_l)$. 其中$l$为数据块的阶数, $\forall 1\leqslant i\leqslant l, \boldsymbol{B}_i\in V_n$.

定义 3. l阶w宽数据切片算法. 数据切片算法$S_{l,w}(\boldsymbol{B}_{1\times l})$以$w$(单位为b)宽度为窗口将$l$阶数据块由顺序存储结构转换为切片存储结构.

当切片窗口宽度w=1 b时, 数据切片算法$S_{l-1}(\boldsymbol{B}_{1\times l})$以1 b为单位将$l$阶顺序存储结构的数据块中的比特完全分离, 每个寄存器字中仅包含1 b来自不同数据分组的相同比特位, 因此适用于通用切片分组密码算法的模型设计. 当w>1时, 此时需要结合分组密码算法的结构特性, 通过分析其基本处理单元, 选择合适的切片窗口宽度, 从而为切片分组密码算法提供合适的数据组织形式.

定义 4. l阶切片分组密码算法. 给定一个六元组$\left(\{V_n\}_l, V_p, E_l, D_l, S_{l,w}, \{V_m\}_l\right)$, $\{V_n\}_l$表示l阶n维明文空间, V_p代表p维密钥空间, E_l代表加密函数, D_l代表解密函数, $S_{l,w}$代表l阶w宽数据切片算法, $\{V_m\}_l$代表l阶m维密文空间. $S_{l,w}\left(E_l\left(S_{l,w}(\boldsymbol{B}_{1\times l}),\boldsymbol{k}\right)\right)\in\{V_m\}_l$表示$l$阶明文块$\boldsymbol{B}_{1\times l}\in\underbrace{V_n\times V_n\times\cdots\times V_n}$经过数据切片算法作用后在密钥$\boldsymbol{k}\in V_p$的作用下经过加密函数处理后, 再次经过数据切片算法作用后都对应一个特定的l阶密文块. $S_{l,w}\left(D_l\left(S_{l,w}(\boldsymbol{C}_{1\times l}),\boldsymbol{k}\right)\right)\in\{V_n\}_l$表示$l$阶密文块$\boldsymbol{C}_{1\times l}\in\underbrace{V_m\times V_m\times\cdots\times V_m}$经过数据切片算法作用后在密钥$\boldsymbol{k}\in V_p$的作用下经过解密函数处理后, 再次经过数据切片算法作用后都对应一个特定的l阶明文块.

在切片分组密码算法中, 每次加密或解密操作会执行2次数据切片算法, 一次是在加密或解密开始之前, 用于将l阶明文块由顺序存储结构转换为切片存储结构; 另一次是在加密或解密结束之后, 用于将l阶密文块恢复为顺序存储结构. 切片分组密码算

法的阶数体现了该算法处理数据块的并行能力，与目标平台处理器指令字长紧密相关. 在 x86 平台上，当利用 64 b 普通指令集实现切片分组密码算法时，切片分组密码算法的阶数最大为 64；而当利用 128 b SSE 扩展指令集时，切片分组密码算法的阶数最大可达 128.

2.2 通用切片分组密码算法设计

由 2.1 节可知，相比分组密码算法以串行方式顺序地处理每个明文分组，切片分组密码算法是对 l 阶数据块进行并行加解密的，因此在相同平台环境下，切片分组密码算法的性能优于分组密码算法. 在 N 位处理器指令集架构下，我们可以将分组密码算法等价转换为切片分组密码算法，转换过程包含明文数据切片、基于切片数据的密码算法设计以及密文数据切片 3 个步骤，具体见图 3.

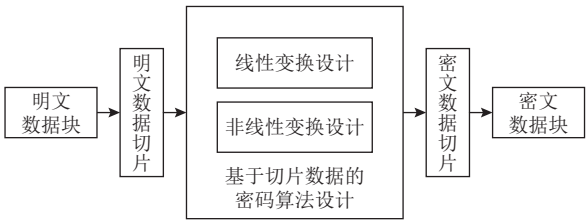


Fig. 3 Design of the general bitsliced block cipher algorithm

图 3 通用切片分组密码算法设计

1）明文数据切片

比特切片技术要求处理器以比特为粒度对数据进行操作. 明文数据切片步骤利用数据切片算法将明文数据块由顺序存储结构转换为切片存储结构，使得不同明文分组的位于相同位置的比特集中在同一个寄存器字中，从而在寄存器级别同时操作所有明文分组.

2）基于切片数据的密码算法设计

比特之间的操作包含逻辑与、逻辑或、逻辑非、逻辑异或等基本逻辑位运算，因此，当把数据组织形式转换为切片存储结构后，需要相应地将分组密码算法中的线性变换和非线性变换运算等价表示为比特之间的逻辑运算. 特别地，当密码算法中包含的非线性代替查表操作时，需要将查找表等价转换为布尔函数，从而使用基本逻辑运算实现查找表.

3）密文数据切片

当完成比特粒度的数据加密处理后，此时密文数据组织形式为切片存储结构，需要对密文数据块施加与明文数据块相同的数据切片操作，将数据的组织形式恢复为顺序存储结构，以正确显示密文数据块.

由图 3 可知，当将分组密码算法设计为切片分组密码算法时，会在加密算法前后引入 2 个在顺序存储结构和切片存储结构之间互相转换的数据切片操作. 为了减小数据转换所带来的额外计算开销，必须设计高效的数据切片算法.

数据切片算法中包含 2 个参数：数据块阶数 l 和切片窗口宽度 w，如何确定两者所存在的依赖关系呢？当在 N 位处理器执行 l 阶 w 宽数据切片算法时，l 阶数据块共占用 $l\times n/N$ 个存储字. 由于以 w 为切片窗口宽度对数据块进行切片，因此切片完成后每个存储字包含 w 位来自不同数据分组的相同比特位，则切片数据块共占用 n/w 个存储字. 数据切片算法仅将数据块在顺序存储结构和切片存储结构之间互相转换，不改变数据块所占存储空间的大小，因此有如下关系成立：

$$l\times n/N = n/w. \tag{3}$$

由式(3)可得 $l=N/w$. 由此可知，数据切片算法只与数据块阶数 l、切片窗口宽度 w 和处理器数据位宽 N 有关，而与分组密码算法的分组长度 n 无关. 当切片窗口宽度 w=1 时，此时切片分组密码的阶数 l 等于处理器所能操作的最大数据位宽 N，由于每个存储字中仅包含 1 b 来自不同数据分组的相同比特位，因此适用于通用分组密码算法的切片化.

图 4 展示了 N 阶明文块在 N 位处理器上的顺序存储结构，其中 $b_{i,j}$ 表示第 i 个明文分组的第 j 个位. 图 5 是明文块经由 N 阶 1 b 宽数据切片算法处理后的切片明文块. 由图 5 可知，切片完成后的第 1 个存储字集中存储了所有 N 个明文分组的第 1 个位，第 2 个存储字集中存储了所有 N 个明文分组的第 2 个位，以此类推. 当对 N 阶切片明文块进行加密处理时，N

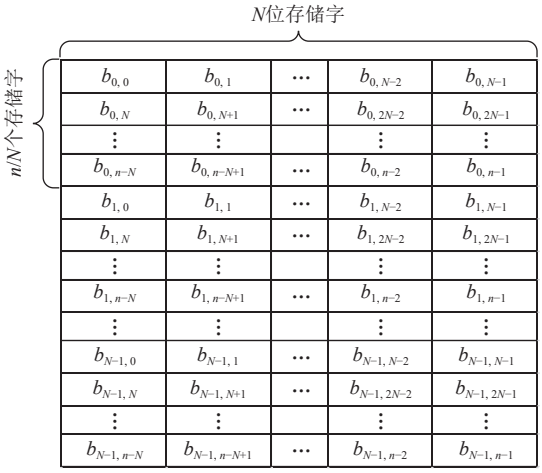


Fig. 4 Sequential storage structure for plaintext bundle

图 4 明文块顺序存储结构

位处理器作为 SIMD 机器并行执行 N 个 1 b 指令运算. 因此, 在 N 位处理器上对切片明文块进行逻辑运算时, 可以同时对 N 个明文分组进行运算.

N位存储字

$b_{N-1,0}$	$b_{N-2,0}$	⋯	$b_{1,0}$	$b_{0,0}$
$b_{N-1,1}$	$b_{N-2,1}$	⋯	$b_{1,1}$	$b_{0,1}$
⋮	⋮		⋮	⋮
$b_{N-1,n/N-1}$	$b_{N-2,n/N-1}$	⋯	$b_{1,n/N-1}$	$b_{0,n/N-1}$
$b_{N-1,n/N}$	$b_{N-2,n/N}$	⋯	$b_{1,n/N}$	$b_{0,n/N}$
$b_{N-1,n/N+1}$	$b_{N-2,n/N+1}$	⋯	$b_{1,n/N+1}$	$b_{0,n/N+1}$
⋮	⋮		⋮	⋮
$b_{N-1,2n/N-1}$	$b_{N-2,2n/N-1}$	⋯	$b_{1,2n/N-1}$	$b_{0,2n/N-1}$
⋮	⋮		⋮	⋮
$b_{N-1,n(N-1)/N}$	$b_{N-2,n(N-1)/N}$	⋯	$b_{1,n(N-1)/N}$	$b_{0,n(N-1)/N}$
$b_{N-1,n(N-1)/N+1}$	$b_{N-2,n(N-1)/N+1}$	⋯	$b_{1,n(N-1)/N+1}$	$b_{0,n(N-1)/N+1}$
⋮	⋮		⋮	⋮
$b_{N-1,n-1}$	$b_{N-2,n-1}$	⋯	$b_{1,n-1}$	$b_{0,n-1}$

Fig. 5 Bitsliced storage structure for plaintext bundle

图 5 明文块切片存储结构

在 N 位处理器上将分组密码算法设计为通用切片分组密码算法的转换开销是计算 N 阶数据块的 2 次数据存储格式转换. 因此, 使用有效的对 $N\times n$ 位数据块进行转换的切片算法对提升通用切片分组密码算法的性能至关重要. 我们在 $N\times N$ 位数据块切片算法的研究基础上[12,19]提出了一种支持大小为 $N\times n$ 位数据块的扩展比特切片算法 *ExtendedBinaryMatrixTranspose*(算法 1). 算法 1 首先将 N 个明文分组在内存中具有相同偏移位置的存储字聚集成为一个大小为 $N\times N$ 位的数据块, 然后调用比特切片算法 *BinaryMatrixTranspose*(算法 2)完成该偏移部分的数据块的转置. 重复上述过程, 当偏移位置为 $n/N-1$ 的 $N\times N$ 位数据块完成数据切片时, 整个 $N\times n$ 位数据块也就完成了切片过程. 在算法 1 中, 外层循环时间复杂度为 $O(n/N)$, 内层循环时间复杂度为 $O(N)$, 内层 $N\times N$ 位切片算法复杂度为 $O(N/2\times \mathrm{lb}(N))$. 因此算法 1 的整体时间复杂度为 $O(n+n/2\times \mathrm{lb}(N))$, 能够高效完成 $N\times n$ 位数据块的切片过程.

算法 1. 扩展比特切片算法.

输入: 待切片的比特数组 $\boldsymbol{A}[N][n]$, 存储切片后的比特数组 $\boldsymbol{B}[N][n]$;

输出: 切片后的比特数组 $\boldsymbol{B}[N][n]$.

① for off in $range(0,n/N)$

② for ind in $range(0,N)$

③ $\boldsymbol{B}\left[off\times N+ind\right]=\boldsymbol{A}[ind\times(n/N)+off]$;

④ end for

⑤ $BinaryMatrixTranspose(\boldsymbol{B}[off\times N:off\times(N+1)])$;

⑥ end for

算法 2. 比特切片算法.

输入: 待切片的比特数组 $\boldsymbol{A}[N][N]$;

输出: 切片后的比特数组 $\boldsymbol{A}[N][N]$.

① $m_0\leftarrow(55555555\cdots55555555)_{16}$;

② $m_1\leftarrow(33333333\cdots33333333)_{16}$;

③ $m_2\leftarrow(0F0F0F0F\cdots0F0F0F0F)_{16}$;

④ ⋯

⑤ $m_j\leftarrow(\{0\}^{2^j}\{1\}^{2^j})_2$;

⑥ for j in $range(0,\mathrm{lb}(N))$

⑦ $k\leftarrow 1\ll j$;

⑧ $r\leftarrow k-1$;

⑨ for i in $range(0,N/2)$

⑩ $l\leftarrow 2(i-(i\wedge r))+i\wedge r$;

⑪ $temp\leftarrow(A_l\wedge m_j)\oplus((B_{l+k}\wedge m_j)\ll k)$;

⑫ $A_{l+k}\leftarrow(A_{l+k}\wedge\overline{m_j})\oplus(A_l\wedge\overline{m_j})\gg k$;

⑬ $A_l\leftarrow temp$;

⑭ end for

⑮ end for

3 面向 SIMD 指令的细粒度切片 SM4 算法优化

通常 SIMD 指令数据宽度为 128 b, 根据通用切片分组密码算法模型, 当在一个 128 b 处理器上实现切片 SM4 分组密码算法时, 需要切片处理的明文分组多达 128 组, 不仅需要占用 1 KB 的内存大小, 而且算法需持续等待 128 个明文分组的到来, 并对其进行切片后才能正式进行加解密操作, 使得密码算法的启动时间较长. 我们提出了一种新颖的细粒度并行的切片算法, 在 128 b 处理器上只需切片处理 32 个明文分组, 不仅减小了内存占用, 而且大大减少了加密算法的启动时间.

3.1 细粒度并行的数据切片算法设计

在通用数据切片算法中, 为了满足具有不同分组长度的分组密码算法的切片化, 切片窗口宽度 w 被设置为 1. 由于并未考虑分组密码算法的设计结构, 因此存在并行粒度较大的问题, 导致数据转换开销占比较大, 同时增加了密码算法的启动时间. 设计细粒度并行的切片分组密码算法的关键是结合特定分组密码算法的结构特性选择合适的切片窗口宽度.

在分组密码算法加解密过程中, 通常首先会把

分组长度为 n 位的明文输入分解为 n/M 个 M 位的基本操作单元，然后对基本操作单元进行迭代处理. 在密码算法迭代过程中，不同基本操作单元内部位于相同偏移位置的比特执行相同的运算逻辑，这意味着我们可以将执行相同运算逻辑的比特集中存储在同一个寄存器字中，在对寄存器字进行逻辑运算时，就可以同时对所有比特进行相同的逻辑运算. 因此，基本操作单元的个数决定了切片窗口宽度的大小. 当切片窗口宽度大于基本操作单元个数时，经由数据切片算法作用的切片数据块中一定包含来自不同基本操作单元位于不同偏移位置的比特. 由于不同基本操作单元位于不同偏移位置的比特通常执行不同的运算逻辑，因此也就无法将分组密码算法表示为切片分组密码算法，所以切片窗口宽度必须小于或等于基本操作单元个数.

在 SM4 密码算法中，首先将 128 b 明文输入分成 4 个 32 b 基本操作单元，然后对 32 b 基本操作单元进行非线性变换和线性变换，迭代 32 轮后输出 128 b 密文. 因此，在切片 SM4 分组密码算法设计中，切片窗口宽度最大为 4. 由式(3)可知，此时切片分组密码算法的阶数 l 与处理器数据位宽 N 之间的关系为 $l=N/4$，这意味着切片 SM4 密码算法能够并行处理的数据块规模最小为 $N/4$ 个明文分组.

当使用数据位宽为 128 b 的 SIMD 指令集实现最小并行粒度的切片 SM4 算法时，切片 SM4 算法能够并行处理 $128/4=32$ 个明文分组，这 32 个明文分组在内存中的顺序存储结构如图 6 所示. 其中，$b_{i,j}$ 表示第 i 个明文分组的第 j 个位. 为了将这 32 个明文分组表示为合适的切片结构，我们以 32 b 基本操作单元为单位，将每个明文分组切分为 4 段，然后顺序排列所有 32 个明文分组的每一段，接着利用算法 3 对重新排列后的明文块进行切片，如图 7 所示.

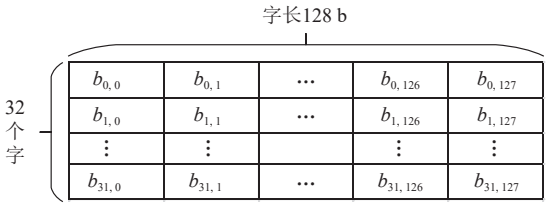


Fig. 6 Sequential storage structure

图 6 顺序存储结构

算法 3 描述了如何将 32 阶数据块在顺序存储结构和以 4 b 为切片窗口宽度的切片存储结构之间互相转换. 与通用数据切片算法不同，在正式开始切片之前，算法 3 首先利用 SIMD 查表指令对每一个明文分组进行字节重排，这一步相当于顺序排列所有 32 个明文分组的每一段. 接着算法 3 对字节重排后的数组进行切片. 切片完成后，切片数组 $\boldsymbol{A}[0:8]$, $\boldsymbol{A}[8:16]$, $\boldsymbol{A}[16:24]$, $\boldsymbol{A}[24:32]$ 分别对应切片明文块 $\boldsymbol{\mathcal{X}}_0,\boldsymbol{\mathcal{X}}_1,\boldsymbol{\mathcal{X}}_2,\boldsymbol{\mathcal{X}}_3$. 其中 $\boldsymbol{\mathcal{X}}_0$ 为所有 32 个明文分组的第 1 个 32 b 基本操作单元的切片存储结构表示，$\boldsymbol{\mathcal{X}}_1$ 为所有 32 个明文分组的第 2 个 32 b 基本操作单元的切片存储结构表示，以此类推.

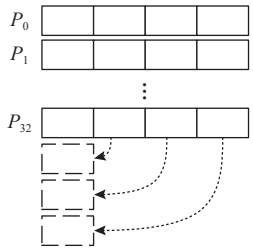


Fig. 7 Slicing structure designed based on general slicing algorithm

图 7 基于通用切片算法设计的切片结构

算法 3 需要调用 2 次，1 次是在正式加密之前，对应切片分组密码算法模型中的明文数据切片步骤，另一次是在加密完成之后，对应切片分组密码算法模型中的密文数据切片步骤. 如果待加密的明文消息长度大于 32 个明文分组，则可以修改切片算法以同时对前一个 32 阶密文块和后一个 32 阶明文块进行切片，进一步降低切片操作在加密算法中的开销占比.

算法 3. 数据切片算法.

输入：明文数据块字节数组 $\boldsymbol{A}[32][16]$;

输出：明文数据块切片数组 $\boldsymbol{A}[32][16]$.

① $index \leftarrow \{\mathrm{F,B,7,3,E,A,6,2,D,9,5,1,C,8,4,0}\}_{16}$;

② $m_0 \leftarrow (55555555\cdots55555555)_{16}$;

③ $m_1 \leftarrow (33333333\cdots33333333)_{16}$;

④ $m_2 \leftarrow (0\mathrm{F0F0F0F}\cdots0\mathrm{F0F0F0F})_{16}$;

⑤ ⋯

⑥ $m_4 \leftarrow (0000\mathrm{FFFF}\cdots0000\mathrm{FFFF})_{16}$;

⑦ for i in $range(0,32)$

⑧ $\quad \boldsymbol{A}[i] = TBL(\boldsymbol{A}[i], index)$;

⑨ end for

⑩ for j in $range(0,5)$

⑪ $\quad k \leftarrow SHL(1,j)$;

⑫ $\quad$ for i in $range(0,16)$

⑬ $\quad\quad r \leftarrow SHL(i-i\%k,1)+i\%k$;

⑭ $\quad\quad temp \leftarrow USHR(\boldsymbol{A}[r+k],k)$;

⑮ $\quad\quad temp \leftarrow XOR(temp,\boldsymbol{A}[r])$;

⑯ $temp \leftarrow AND(temp, m_j)$;

⑰ $\boldsymbol{A}[r] \leftarrow XOR(\boldsymbol{A}[r], temp)$;

⑱ $temp \leftarrow SHL(temp, k)$;

⑲ $\boldsymbol{A}[r+k] \leftarrow XOR(\boldsymbol{A}[r+k], temp)$;

⑳ end for

㉑ end for

在切片 SM4 算法中, 由于明文块由顺序存储结构转换为切片存储结构, 因此为了与明文切片数据相异或, 需要将经由密钥扩展算法生成的子密钥也表示成密钥切片数据, 其切片结构如图 8 所示. 其中, $rk_{i,j}$ 表示第 i 轮子密钥的第 j 个位.

$rk_{i,0}\cdots rk_{i,0}$	$rk_{i,8}\cdots rk_{i,8}$	$rk_{i,16}\cdots rk_{i,16}$	$rk_{i,24}\cdots rk_{i,24}$
$rk_{i,1}\cdots rk_{i,1}$	$rk_{i,9}\cdots rk_{i,9}$	$rk_{i,17}\cdots rk_{i,17}$	$rk_{i,25}\cdots rk_{i,25}$
⋮	⋮	⋮	⋮
$rk_{i,7}\cdots rk_{i,7}$	$rk_{i,15}\cdots rk_{i,15}$	$rk_{i,23}\cdots rk_{i,23}$	$rk_{i,31}\cdots rk_{i,31}$

Fig. 8 Bitsliced storage structure of the round key

图 8 轮密钥切片存储结构

算法 4 描述了如何对轮密钥进行切片. 相比于对明文数据进行切片, 轮密钥切片相对简单. 首先利用 SIMD 指令将大小为 32 b 的轮密钥扩充为 128 b 的数据 ek, 然后利用 SIMD 查表指令对 ek 中的字节进行重排, 接着分别使用无符号左移指令和有符号右移指令对 ek 进行操作, 并将其结果存储在输出数组 K 中, 由此便得到了切片后的轮密钥数组 K.

算法 4. 轮密钥切片算法.

输入: 轮密钥 rk_i;

输出: 切片字节数组 $\boldsymbol{K}$ [8][16].

① $index \leftarrow \{\mathrm{F,B,7,3,E,A,6,2,D,9,5,1,C,8,4,0}\}_{16}$;

② $ek \leftarrow DUP(rk_i)$;

③ $ek \leftarrow TBL(ek, index)$;

④ for i in $range(0,8)$

⑤ $temp \leftarrow SHL(ek, i)$; /* 无符号左移 */

⑥ $\boldsymbol{K}[i] \leftarrow SSHR(temp, 31)$; /*有符号右移 */

⑦ end for

3.2 基于切片数据的线性变换优化设计

在明文和轮密钥的切片形式表示下, 我们需要将 SM4 算法轮函数中的非线性变换和线性变换转换为一系列的逻辑操作, 然后使用处理器指令集中的逻辑指令实现逻辑操作. 本节介绍如何将线性变换表示为以切片数据为输入的逻辑操作.

根据式(2), 线性变换 L 由循环左移和异或运算组成. 设单位矩阵 $\boldsymbol{I}$ 大小为 32×32, 令 $\boldsymbol{I}_c$ 表示将单位矩阵 $\boldsymbol{I}$ 按行向上循环移动 c 行, 则可将式(2)等价表示为

$$z = L(\boldsymbol{y}) = \boldsymbol{I}\cdot\boldsymbol{y} + \boldsymbol{I}_2\cdot\boldsymbol{y} + \boldsymbol{I}_{10}\cdot\boldsymbol{y} + \boldsymbol{I}_{18}\cdot\boldsymbol{y} + \boldsymbol{I}_{24}\cdot\boldsymbol{y} = (\boldsymbol{I} + \boldsymbol{I}_2 + \boldsymbol{I}_{10} + \boldsymbol{I}_{18} + \boldsymbol{I}_{24})\cdot\boldsymbol{y} = \boldsymbol{M}\cdot\boldsymbol{y}, \tag{4}$$

其中矩阵 $\boldsymbol{M} = \boldsymbol{I} + \boldsymbol{I}_2 + \boldsymbol{I}_{10} + \boldsymbol{I}_{18} + \boldsymbol{I}_{24}$. 将矩阵 $\boldsymbol{M}$ 表示为 4×4 大小的分块矩阵, 则有

$$\boldsymbol{M} = \begin{pmatrix} \boldsymbol{m}_0 & \boldsymbol{m}_1 & \boldsymbol{m}_1 & \boldsymbol{m}_2 \\ \boldsymbol{m}_2 & \boldsymbol{m}_0 & \boldsymbol{m}_1 & \boldsymbol{m}_1 \\ \boldsymbol{m}_1 & \boldsymbol{m}_2 & \boldsymbol{m}_0 & \boldsymbol{m}_1 \\ \boldsymbol{m}_1 & \boldsymbol{m}_1 & \boldsymbol{m}_2 & \boldsymbol{m}_0 \end{pmatrix},$$

其中 $\boldsymbol{m}_0, \boldsymbol{m}_1, \boldsymbol{m}_2$ 的具体参数值为

$$\boldsymbol{m}_0 = \begin{pmatrix} 0&1&0&0&0&0&0&1 \\ 1&1&0&1&1&0&0&0 \\ 1&1&0&1&1&0&1&1 \\ 0&0&0&1&0&0&1&1 \\ 0&0&1&1&0&0&0&1 \\ 1&1&0&1&0&1&0&0 \\ 0&1&0&0&0&1&1&1 \\ 1&0&1&1&1&1&0&0 \end{pmatrix},$$

$$\boldsymbol{m}_1 = \begin{pmatrix} 1&0&0&0&0&0&0&0 \\ 0&1&1&0&0&0&1&1 \\ 0&0&0&0&0&0&1&1 \\ 0&1&1&1&0&1&0&0 \\ 0&1&1&0&1&1&0&0 \\ 0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&1&0 \\ 0&0&1&0&0&0&1&0 \end{pmatrix},$$

$$\boldsymbol{m}_2 = \begin{pmatrix} 1&1&0&0&0&0&0&1 \\ 1&0&1&1&1&0&1&1 \\ 1&1&0&1&1&0&0&0 \\ 0&1&1&0&0&1&1&1 \\ 0&1&0&1&1&1&0&1 \\ 1&1&0&1&0&0&0&0 \\ 0&1&0&0&0&1&0&1 \\ 1&0&0&1&1&1&1&0 \end{pmatrix}.$$

线性变换 L 接收非线性变换 τ 的输出作为输入. 令非线性变换 τ 的切片数据输出为 $\boldsymbol{B} = (\boldsymbol{B}_0, \boldsymbol{B}_1, \boldsymbol{B}_2, \boldsymbol{B}_3, \boldsymbol{B}_4, \boldsymbol{B}_5, \boldsymbol{B}_6, \boldsymbol{B}_7)^{\mathrm{T}}$, 其中, $\boldsymbol{B}_i$ 大小为 128 b. 根据式(4)以及矩阵 $\boldsymbol{M}$ 和切片数据的结构, 我们设计了以切片数据作为输入的线性变换公式:

$$L(\boldsymbol{B}) = \boldsymbol{M}\cdot\boldsymbol{B} = \boldsymbol{Q}_0 \oplus (\boldsymbol{Q}_1 \lll 32) \oplus (\boldsymbol{Q}_1 \lll 64) \oplus (\boldsymbol{Q}_2 \lll 96), \tag{5}$$

其中 $\boldsymbol{Q}_0, \boldsymbol{Q}_1, \boldsymbol{Q}_2$ 表示为

$$\begin{cases} \boldsymbol{Q}_0 = \boldsymbol{m}_0 \times \boldsymbol{B}, \\ \boldsymbol{Q}_1 = \boldsymbol{m}_1 \times \boldsymbol{B}, \\ \boldsymbol{Q}_2 = \boldsymbol{m}_2 \times \boldsymbol{B}. \end{cases}$$

式(5)由矩阵乘法、逻辑异或和逻辑移位组成. 其中矩阵乘法是定义在有限域 $GF(2)$ 上的加法运算, 等价于逻辑异或操作. 因此借助式(5)可以使用逻辑异或指令生成以切片结构作为输入的线性变换函数.

3.3 基于切片数据的非线性变换优化设计

非线性变换 τ 由 4 个并行的 S 盒组成, 在切片

SM4 算法中，无法再使用查找表实现 S 盒. 因此需要将 S 盒等价转换为布尔函数. 一种方法是利用 S 盒构造 8 个具有 8 b 输入和 1 b 输出的真值表，然后利用这 8 个真值表综合出 8 个布尔函数，每个布尔函数的输出值代表 S 盒查找表的 1 b 输出位. 但是由于布尔函数的输入变量多达 8 个，因此从真值表中综合出来的布尔函数复杂度较大，所需门函数较多[18]，不适合切片 SM4 算法的高性能实现.

另外一种方法是从 S 盒代数表达式出发，将 S 盒分解为基本的逻辑操作. SM4 算法发布之后，Liu 等人[20] 对 SM4 算法的 S 盒进行分析，给出了 S 盒的代数结构与具体参数值. S 盒的代数表达式为

$$S(\boldsymbol{x}) = \boldsymbol{A} \times INV(\boldsymbol{A} \times \boldsymbol{x} + \boldsymbol{C}) + \boldsymbol{C}, \tag{6}$$

其中求逆运算$INV()$定义在有限域$\{GF(2^8): F(\boldsymbol{x}) = \boldsymbol{x}^8 + \boldsymbol{x}^7 + \boldsymbol{x}^6 + \boldsymbol{x}^5 + \boldsymbol{x}^4 + \boldsymbol{x}^2 + 1\}$上. 矩阵 $\boldsymbol{A}$ 与向量 $\boldsymbol{C}$ 为固定参数值，具体数值请参考文献 [20].

根据式(6)可知，S 盒的计算复杂度主要由有限域$GF(2^8)$上的求逆运算决定. 对于求解有限域上一个元素的逆元素，一种做法是利用扩展欧几里得算法进行求解. 但是对于比较大的有限域，如$GF(2^8)$，求逆函数计算复杂度高、运算速度慢. 借助有限域理论知识，我们可以将有限域$GF(2^8)$上的求逆运算转换到塔域$GF(((2^2)^2)^2)$完成，从而大大降低求逆运算的复杂度.

首先，借助$GF(2^8)$和$GF((2^4)^2)$之间的同构映射矩阵将有限域$GF(2^8)$中的元素表示为有限域$GF(2^4)$上的线性多项式. 然后，使用$GF(2^4)$和$GF(2^2)$可以将有限域$GF(2^4)$上的元素表示为有限域$GF(2^2)$上的线性多项式. 接着，使用$GF(2^2)$和$GF(2)$可以将有限域$GF(2^2)$上的元素表示为有限域$GF(2)$上的线性多项式. 采用经典的塔域结构[21] 优化 S 盒求逆运算：

$$\begin{cases} GF(2^8)/GF(2^4): r(\boldsymbol{y}) = y^2 + \tau y + v = (y + Y^{16})(y + Y), \\ GF(2^4)/GF(2^2): s(z) = z^2 + Tz + N = (z + Z^4)(z + Z), \\ GF(2^2)/GF(2): t(w) = w^2 + w + 1 = (w + W^2)(w + W). \end{cases}$$

最后，经过一系列的有限域变换，可以将 S 盒中的求逆运算递归转换到有限域$GF(2)$上的乘法与加法运算. 而有限域$GF(2)$上的乘法运算和加法运算等价于逻辑与运算和逻辑异或运算，因此可以使用处理器指令集中的逻辑与和逻辑异或指令生成以比特切片结构作为输入的 S 盒求逆函数.

4 实验结果分析

随着移动互联网和智能终端的普及，大量手机应用催生了算力架构的持续演进，能构建更高性能、更低功耗计算平台的 ARM 架构正在成为算力发展的主流. 因此，本文以 ARM 架构平台为测试环境对切片 SM4 密码算法进行性能测试分析. SM4 加密算法和解密算法结构完全相同，唯一不同的是轮密钥的使用顺序相反，因此加密算法和解密算法性能相近. 本文以加密算法的处理性能来比较普通查表 SM4 算法和切片 SM4 算法的性能表现. 此外，切片 SM4 算法能够并行处理 32 组 128 b 的明文数据，当待处理的数据规模小于 512 B 时，需要对其进行数据填充，因此影响切片 SM4 算法性能的一个重要因素是数据大小. 基于上述考虑，本文基于多种处理器对 SM4 比特切片算法在不同输入规模数据下的加密性能进行了测试分析. 表 2 给出了实验环境的硬件配置.

Table 2 Hardware Configure of the Experiment Environment

表 2 实验环境硬件配置

平台	处理器	主频/GHz
FT-1500A/4	FTC660×4	1.5~2
FT-2000/4	FTC663×4	2.2/2.6
Raspberry	Cortex-A72	1.5
Macos	Apple M1	3.2

4.1 切片 SM4 密码算法实现

本节基于第 3 节设计的算法 3，利用 NEON 指令集实现了数据切片算法、基于切片数据的非线性变换和线性变换.

4.1.1 数据切片算法实现

算法 3 和算法 4 以伪代码的形式直观地描述了如何利用通用 SIMD 指令集实现切片过程，因此当利用 NEON 指令集实现数据切片算法时，只需要将伪代码中对应的逻辑伪指令翻译为对应的处理器逻辑指令即可. 算法 3 和算法 4 中所使用到的逻辑伪指令与 NEON 指令集中的逻辑指令的对应关系如表 3 所示，只需进行简单的逻辑指令替换即可快速实现数

Table 3 Corresponding Relation Between Logical Instructions and NEON Instructions

表 3 逻辑指令与 NEON 指令对应关系

逻辑指令	NEON 指令
AND	AND
XOR	EOR
SHL	SHL
USHR	USHR
SSHR	SSHR

据切片算法.

4.1.2 线性变换优化实现

在式(5)中,需要对$\boldsymbol{Q}_0$,$\boldsymbol{Q}_1$,$\boldsymbol{Q}_2$进行循环左移,注意到$\boldsymbol{Q}_0$,$\boldsymbol{Q}_1$,$\boldsymbol{Q}_2$是具有 8 个 128 b 分量的向量,因此需要分别循环左移其中的每个分量. 但是,NEON 指令集并未提供针对 128 b 的左移指令,我们使用字节重排指令 EXT 等价实现了针对 128 b 的循环左移运算. 实现线性变换所需的相关指令统计如表 4 所示.

Table 4 Logical Instructions Statistics Required for Linear Transform

表 4 线性变换所需的逻辑指令统计

CPU 指令	指令个数
EOR	45
MVN	3
EXT	24
总计	72

4.1.3 非线性变换优化实现

Käspe 等人[14]基于 Canright 等人[21]的研究成果,将逻辑运算等价转换为 x86 平台上的 SSE 指令集,仅使用了 163 条相关指令便完成了塔域$GF(((2^2)^2)^2)$上的求逆运算. 本文在此基础上,通过参考 Linux 内核与 OpenSSL 中对 AES 的 S 盒的处理,对非线性变换进行 NEON 汇编指令粒度的优化,仅用 104 条逻辑指令便完成了塔域$GF(((2^2)^2)^2)$上的求逆运算,相关指令统计如表 5 所示.

Table 5 Logical Instructions Statistics Required for Nonlinear Transform

表 5 非线性变换所需的逻辑指令统计

CPU 指令	指令个数
EOR	67
AND	28
OR	4
MVN	1
BSL	4
总计	104

注意到表 5 中出现了 2 条指令: MVN 和 BSL. 其中, MVN 指令是 NEON 指令集提供的用于计算逻辑非运算的指令; 而 BSL 是一个三目运算符,其运算规则为

$$BSL(a,b,c) = c \text{ XOR } ((c \text{ XOR } b) \text{ AND } a).$$

4.2 大规模短数据性能测试

本文首先构造了数据长度从 16~8 192 B 不等的若干组测试数据,每组测试数据包含 100 000 条短数据. 然后在实验平台上比较通用 SM4 算法和切片 SM4 算法对大规模短数据的加密效果,测试结果如图 9~12 所示.

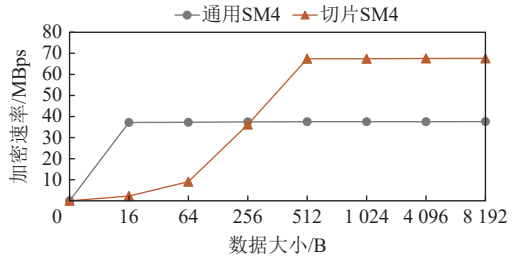


Fig. 9 Short data encryption performance for FT-1500A/4

图 9 FT-1500A/4 短数据加密性能

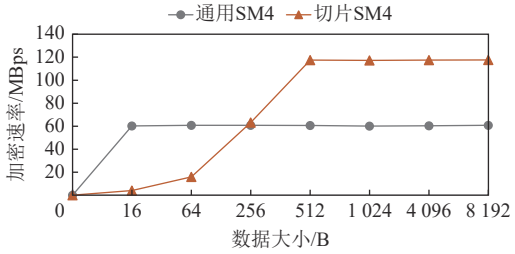


Fig. 10 Short data encryption performance for FT-2000/4

图 10 FT-2000/4 短数据加密性能

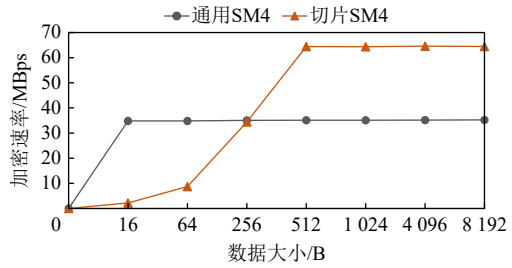


Fig. 11 Short data encryption performance for raspberry

图 11 树莓派短数据加密性能

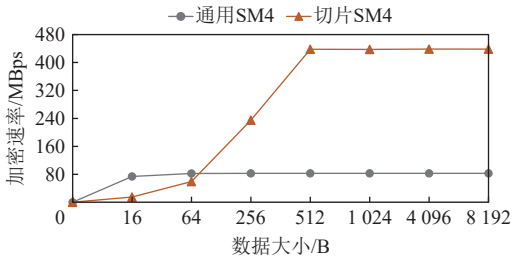


Fig. 12 Short data encryption performance for Macos

图 12 Macos 短数据加密性能

由图 9~12 可知,通用 SM4 算法的加密速率保持恒定,不随数据规模的改变而变化. 不过对于切片 SM4 算法,当数据长度小于 512 B 时,其加密速率随数据规模的增加而变大,直到数据长度超过 512 B 时,

加密效率才保持恒定不变；此外当数据规模小于 256 B 时，切片 SM4 算法的加密性能低于通用 SM4 算法. 究其原因，切片 SM4 算法是对 32 组 128 b 大小的数据块并行处理的，当明文消息长度小于 512 B 时，切片 SM4 算法需要对明文消息进行填充，然后再对填充后的消息进行加密处理. 因此当明文数据较短时，切片 SM4 算法的并行处理优势无法抵消对数据进行填充带来的额外开销，所以才出现了性能不及通用 SM4 算法的情况. 随着明文消息长度的增加，对数据进行填充带来的额外开销越来越小，切片 SM4 算法的性能快速提升，远超通用 SM4 算法.

4.3 大规模长数据性能测试

为了探究 SM4 比特切片算法对大规模长数据的性能表现，本文接着构造了若干组数据长度在 20~100 MB 的测试数据，每组测试数据包含 100 000 条长数据，并对其进行了测试分析. 测试结果如图 13~16 所示.

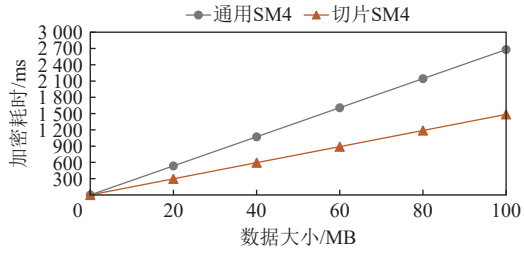


Fig. 13 Long data encryption performance for FT-1500A/4

图 13 FT-1500A/4 长数据加密性能

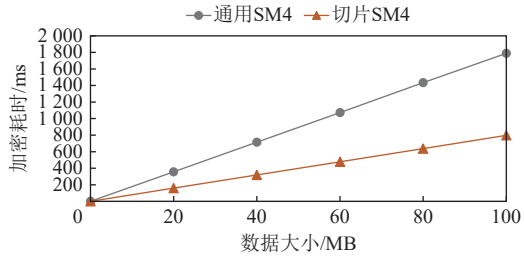


Fig. 14 Long data encryption performance for FT-2000/4

图 14 FT-2000/4 长数据加密性能

由图 13~16 可知，无论是通用 SM4 算法还是切片 SM4 算法，两者对大规模数据的处理时间均随数据规模的增加而线性增加；在对相同规模的数据进行加密处理时，相比通用 SM4 算法，切片 SM4 算法在 FT-1500A/4 处理器上的加密速率可达 67.6 MBps，加密耗时平均下降了 44.6%，加密性能平均提升了 80.4%；而在 FT-2000/4 处理器上的加密速率则高达 117.6 MBps，加密耗时平均下降了 47.7%，性能平均提升了 91.2%. 切片 SM4 算法在树莓派上的表现和 FT-1500A/4 类似，而在 Macos 上的表现远远超过通用 SM4 算法，其加密速率最高可达 438.0 MBps，加密耗时平均下降了 81.1%，加密性能平均提升 430.3%. 这表明比特切片算法带来的数据并行处理优势非常明显，不仅能够加速大规模短数据的加密性能，同样适用于对大规模长数据进行批量加密.

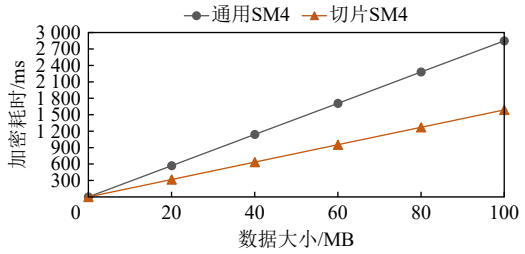


Fig. 15 Long data encryption performance for raspberry

图 15 树莓派长数据加密性能

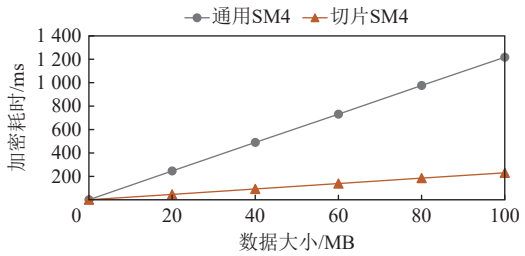


Fig. 16 Long data encryption performance for Macos

图 16 Macos 长数据加密性能

4.4 与其他优化算法对比

Zhang 等人[17] 提出了一种以 1 b 为切片窗口宽度的切片 SM4 算法，并分别在 x86 处理器和 Contex-A7 处理器上进行了测试分析. Kwon 等人[10] 则提出了一种基于 SIMD 查表的 SM4 实现方法，并在 Apple A13 Bionic 处理器上进行了性能测试分析. 为了公平地与这些优化算法进行性能对比分析，我们按照文献 [10] 中描述的技术思想，分别实现了 Zhang 等人[17] 的切片 SM4 算法和 Kwon 等人[10] 的通用 SM4 算法，并在 FT-2000/4 和 Macos 平台上进行了性能测试. 为了消除 CPU 主频对密码算法性能的影响，本文采用 CPU 主频和加密速率的比值，即加密每字节所需的时钟周期作为衡量指标，实验结果如图 17 所示.

在 FT-2000/4 平台上，本文提出的切片 SM4 算法加密性能达到了 20.6 CPB，相比 Zhang 等人[17] 提出的切片 SM4 算法性能提升了 51.5%，相比 Kown 等人[10] 提出的通用 SM4 算法性能提升了 99.0%. 而在 Macos 平台上，本文提出的切片 SM4 算法加密性能高达 7.0 CPB，相比 Zhang 等人[17] 和 Kown 等人[10] 提出的 SM4 优化算法性能分别提升了 64.3% 和 110.0%. 这

表明本文提出的细粒度并行的切片 SM4 算法具有很大的性能优势. 究其原因, 本文设计了更加细粒度的数据切片算法, 并利用 NEON 指令集完成了明文和子密钥的高效切片操作, 大大降低了加密算法的启动时间, 充分利用了比特切片技术的性能潜能.

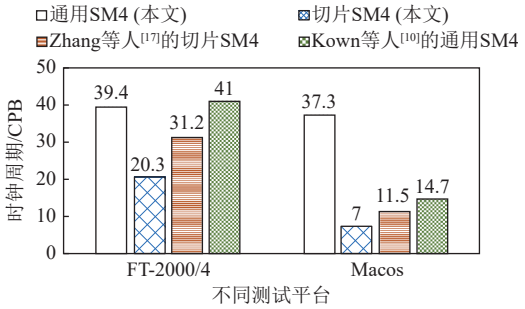


Fig. 17 Comparison of our algorithm and other optimized algorithms

图 17 本文和其他优化算法对比

5 结 论

本文通过对分组密码算法和处理器指令集进行建模, 提出了一种通用的切片分组密码算法模型. 在此基础上, 针对 SM4 算法结构特性, 设计了一种细粒度并行的切片 SM4 密码算法, 能够并行处理 32 组 128 b 明文分组, 并利用 NEON 指令集优化实现了切片 SM4 密码算法, 能够抵御针对缓存的计时攻击等侧信道攻击. 实验结果表明, 本文提出的切片 SM4 密码算法的加密性能相比通用 SM4 算法平均提升了 80.4%~430.3%, 相比其他优化算法性能也有大幅度的提升, 适用于对密码性能要求较高的资源有限的设备.

作者贡献声明: 王闯提出了算法思路和实验方案并撰写论文; 丁滟提出指导意见并修改论文; 黄辰林提出指导意见并修改论文; 宋连涛提供实验环境并负责部分实验测试.

参 考 文 献

[1] Schuster F, Costa M, Fournet C, et al. VC3: Trustworthy data analytics in the cloud using SGX[C]//Proc of the 31st IEEE Symp on Security and Privacy. Piscataway, NJ: IEEE, 2015: 38−54

[2] Dinh T T A, Saxena P, Chang E C, et al. M2R: Enabling stronger privacy in MapReduce computation[C]//Proc of the 24th USENIX Security Symp. Berkeley, CA: USENIX Association , 2015: 447−462

[3] Cao Zhenfu, Dong Xiaolei, Zhou Jun, et al. Research advances on big data security and privacy preserving[J]. Journal of Computer Research and Development, 2016, 53(10): 2137−2151(in Chinese)
(曹珍富, 董晓蕾, 周俊, 等. 大数据安全与隐私保护研究进展[J]. 计算机研究与发展, 2016, 53(10): 2137−2151)

[4] Lü Shuwang, Li Dawei, Zhang Chao, et al. GM/T0002—2012 SM4 block cipher algorithm[S]. Beijing: China Standard Press, 2012 (in Chinese)
(吕述望, 李大为, 张超, 等. GM/T0002—2012 SM4 分组密码算法[S]. 北京: 中国标准出版社, 2012)

[5] Gao Xianwei, Lu Erhong, Xian Liqin, et al. FPGA implementation of the SMS4 block cipher in the Chinese WAPI standard[C]//Proc of the 4th Int Conf on Embedded Software and Systems Symp. Piscataway, NJ: IEEE, 2008: 104−106

[6] Jin Yier, Shen Haibin, You Rongquan. Implementation of SMS4 block cipher on FPGA[C]//Proc of the 1st Int Conf on Communications and Networking in China. Piscataway, NJ: IEEE, 2006: 1−4

[7] Guan Zhenyu, Li Yunhao, Shang Tao, et al. Implementation of SM4 on FPGA: Trade-off analysis between area and speed[C]//Proc of the 1st IEEE Int Conf on Intelligence and Safety for Robotics. Piscataway, NJ: IEEE, 2018: 192−197

[8] He Shiyang, Li Hui, Li Fenghua. Optimization and implementation of the SM4 on FPGA[J]. Journal of Xidian University, 2021, 48(3): 155−162 (in Chinese)
(何诗洋, 李晖, 李凤华. SM4 算法的 FPGA 优化实现方法[J]. 西安电子科技大学学报, 2021, 48(3): 155−162)

[9] Lang Huan, Zhang Lei, Wu Wenling. Fast software implementation of SM4[J]. Journal of University of Chinese Academy of Sciences, 2018, 35(2): 180−187 (in Chinese)
(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)

[10] Kwon H, Kim H, Eum S, et al. Optimized implementation of SM4 on AVR microcontrollers, RISC-V mrocessors, and ARM processors[J]. IEEE Access, 2022, 10: 80225−80233

[11] Biham E. A fast new DES implementation in software[C]//Proc of the 4th Int Workshop on Fast Software Encryption. Berlin: Springer, 1997: 260−272

[12] Rebeiro C, Selvakumar D, Devi A S L. Bitslice implementation of AES[C]//Proc of the 5th Int Conf on Cryptology and Network Security. Berlin: Springer, 2006: 203−212

[13] Könighofer R. A fast and cache-timing resistant implementation of the AES[C]//Proc of the 8th Cryptographers' Track at the RSA Conf. Berlin: Springer, 2008: 187−202

[14] Käsper E, Schwabe P. Faster and timing-attack resistant AES-GCM[C/OL]//Proc of the 11th Int Workshop on Cryptographic Hardware and Embedded Systems. Berlin: Springer, 2009[2022-11-09].https://dl-acm-org-s.libyc.nudt.edu.cn/doi/10.1007/978-3-642-04138-9_1

[15] Scheibelhofer K. A bit-slice implementation of the whirlpool Hash function[C]//Proc of the 7th Cryptographers' Track at the RSA Conf. Berlin: Springer, 2007: 385−401

[16] Matsui M, Nakajima J. On the power of bitslice implementation on Intel Core2 processor[C]//Proc of the 9th Int Workshop on Cryptographic Hardware and Embedded Systems. Berlin: Springer, 2007: 121−134

[17] Zhang Jingbin, Ma Meng, Wang Ping. Fast implementation for SM4 cipher algorithm based on bit-slice technology[C]//Proc of the 3rd Int

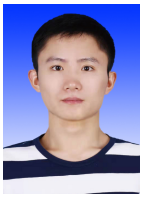
Conf on Smart Computing and Communication. Berlin: Springer, 2018: 104−113

[18] Zhang Xiaocong, Guo Hua, Zhang Xiyong, et al. Fast software implementation of SM4[J]. Journal of Cryptologic Research, 2020, 7(6): 799−811(in Chinese)
(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)

[19] Gaubatz G, Sunar B. Leveraging the multiprocessing capabilities of modern network processors for cryptographic acceleration[C]//Proc of the 4th IEEE Int Symp on Network Computing and Applications. Piscataway, NJ: IEEE, 2005: 235−238

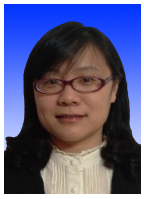
[20] Liu Fen, Ji Wen, Hu Lei, et al. Analysis of the SMS4 block cipher[C]//Proc of the 12th Australasian Conf on Information Security and Privacy. Berlin: Springer, 2007: 158−170

[21] Canright D, Batina L. A very compact "perfectly masked" S-box for AES[C]//Proc of the 6th Int Conf on Applied Cryptography and Network Security. Berlin: Springer, 2008: 446−459



Wang Chuang, born in 1997. Master. His main research interests include smart contract of block-chain and cryptography algorithm optimization.

王　闯, 1997 年生. 硕士. 主要研究方向为区块链智能合约、密码算法优化.



Ding Yan, born in 1977. PhD, associate professor. Distinguished member of CCF. Her main research interests include operating system, system security, and trusted cloud computing.

丁　滟, 1977 年生. 博士, 副研究员. CCF 杰出会员. 主要研究方向为操作系统、系统安全、可信云计算.



Huang Chenlin, born in 1976. PhD, professor. Member of CCF. His main research interests include operating system, system security, and smart security. (clhuang@nudt.edu.cn)

黄辰林, 1976 年生. 博士, 研究员. CCF 会员. 主要研究方向为操作系统、系统安全、智能安全.



Song Liantao, born in 1998. PhD candidate. His main research interests include system security and trusted computing. (songliantao@nudt.edu.cn)

宋连涛, 1998 年生. 博士研究生. 主要研究方向为系统安全、可信计算.