密码学报 ISSN 2095-7025 CN 10-1195/TN
Journal of Cryptologic Research, 2021, 8(3): 388–401

E-mail: jcr@cacrnet.org.cn
http://www.jcr.cacrnet.org.cn
Tel/Fax: +86-10-82789618

ZUC 算法软件快速实现*

张宇鹏[1], 高 莹[1,2], 严 宇[1], 刘 翔[1]

1. 北京航空航天大学 网络空间安全学院, 北京 100191
2. 空天网络安全工业和信息化部重点实验室, 北京 100191

通信作者: 高莹, E-mail: gaoying@buaa.edu.cn

摘 要: ZUC 算法, 即 ZUC-128 流密码算法, 是首个成为国际商业密码标准的国产密码算法. 目前, ZUC-128 算法和后续 ZUC-256 算法的硬件优化工作卓有成效, 其 IP 核的运行速度可以达到 100 Gbps. 但对 ZUC 算法软件实现速度的研究一直比较缓慢, 相关研究不多. 为优化 ZUC 算法的软件实现速度, 我们尝试了 8 种软件优化方式. 通过实验分析得知, 使用多线程并行的方式优化 ZUC 算法反而会降低算法的运行效率, 且有些方法单独使用有效但与其他方法组合之后算法的实现速度不增反减. 因此需进行优化组合, 通过实验最终选择了顺序组合: (1) 使用优化函数的调用过程; (2) 编译器优化; (3) 延迟取模; (4) 合并 S 盒, 探索出了一种 ZUC 算法软件优化方法的高效组合方式. 利用这种高效组合方式, 在 Intel Core i7-8750H@2.20 GHz 处理器上, 生成长密钥流时, ZUC 算法的软件实现速度达到 4.22 Gbps. 和已有的最新结果 3.34 Gbps 相比速度提高 26%, 本方法同样适用于 ZUC-256 流密码算法的软件提速.

关键词: ZUC 算法; 软件优化; 流密码; SIMD

中图分类号: TP309.7 **文献标识码:** A DOI: 10.13868/j.cnki.jcr.000446

中文引用格式: 张宇鹏, 高莹, 严宇, 刘翔. ZUC 算法软件快速实现[J]. 密码学报, 2021, 8(3): 388–401. [DOI: 10.13868/j.cnki.jcr.000446]

英文引用格式: ZHANG Y P, GAO Y, YAN Y, LIU X. Fast software implementation of ZUC algorithm[J]. Journal of Cryptologic Research, 2021, 8(3): 388–401. [DOI: 10.13868/j.cnki.jcr.000446]

Fast Software Implementation of ZUC Algorithm

ZHANG Yu-Peng[1], GAO Ying[1,2], YAN Yu[1], LIU Xiang[1]

1. School of Cyber Science and Technology, Beihang University, Beijing 100191, China
2. Key Laboratory of Aerospace Network Security, Ministry of Industry and Information Technology, Beijing 100191, China

Corresponding author: GAO Ying, E-mail: gaoying@buaa.edu.cn

Abstract: ZUC algorithm (i.e., ZUC-128 stream cipher) is a standard encryption algorithm for LTE. Up to now, the hardware optimization work of the ZUC-128 algorithm (similarly for ZUC-256) has been fruitful, and the operating speed of its IP core can reach 100 Gbps. However, the software implementation speed of ZUC algorithm has been slow, and there is not much related research. To optimize the software implementation of ZUC algorithm, eight methods are tried. Through experimental

* 基金项目: 北京市自然科学基金 (M21033); 国家自然科学基金 (61932011, 61972017)
Foundation: Natural Science Foundation of Beijing Municipality (M21033); National Natural Science Foundation of China (61932011, 61972017)
收稿日期: 2020-07-07 定稿日期: 2020-09-20

analysis, it is observed that using multi-thread parallel optimization of ZUC algorithm will slow down and reduce the efficiency of the algorithm, and some methods are effective when they are used alone. However, when the methods are combined together, the implementation speed of the algorithm does not increase but decrease. This paper explores an efficient software optimization sequential combination method of ZUC algorithm through experiments: (1) optimizing the calling process of functions; (2) compiler optimization; (3) modular delay; (4) merging S-boxes. Using this efficient combination method, on the Intel Core i7-8750H@2.20 GHz processor, software implementation speed of ZUC-128 algorithm achieves 4.22 Gbps when generating long cipher stream. Compared with the latest result of 3.34 Gbps, the speed increased by 26%. In addition, this method can be fully applied to ZUC-256 stream cipher.

Key words: ZUC algorithm; software optimization; stream ciphers; SIMD

1 引言

ZUC 算法 (即 ZUC-128 序列密码算法) 是我国自主设计的密码算法, 也是我国第一个成为国际商业密码标准的国产密码算法. 2011 年 9 月, 在第三代合作伙伴计划 (3GPP) 组织的第 53 次会议上, 基于 ZUC 密码算法的数据加密算法 EEA3 和认证算法 EIA3 成为 3GPP LTE 标准 [1]. 2020 年 4 月, 在国际标准化组织 ISO 的工作组会议上, ZUC 算法成为 ISO/IEC 国际标准 [2], 进入标准发布阶段. ZUC 算法是我国商用密码的重要组成部分, 主要用于通信领域的数据机密性和完整性保护工作. 2018 年, 为了适应 5G 通信的需求, ZUC 算法研制组研制了 “ZUC-256 流密码算法”, 从 128 比特密钥升级到 256 比特密钥, 升级版的算法与 ZUC-128 算法高度兼容, 可提供消息加密和认证 [3].

在优化 ZUC 算法实现、提高工作效率方面, 国内外学者已经在软件和硬件方面做出了很多尝试. 总体来说, 在 ZUC 算法的硬件实现方面取得的速度令人满意, Zhang 等人 [4] 使用一种新的展开架构将 ZUC 算法在 ASIC 上的实现速度提高到 100 Gbps. 但是, 由于 ZUC 算法的设计结构所限, ZUC 算法在软件实现速度远远不如硬件速度. 在 ZUC 算法标准报告 [5] 中, 提到了一种利用移位和与运算实现模 $2^{31}-1$ 的加法和乘法的快速运算方法. Avanzi 等人 [6] 提出用延迟取模的方法提高 LFSR 的实现效率. 李建鹏等人 [7] 使用优化函数调用过程和优化执行过程的方法, 减少了访存时的冲突, 并减少了运行时取模运算的次数. Yu 等人 [8] 使用延迟取模、滑动窗口和将 S 盒合并的方式, 结合使用 SIMD 指令集, 将 ZUC 算法的软件实现速度提高到 3.34 Gbps. Drucker 等人 [9] 利用了 S 盒和 AES 密码中的 S 盒的同构, 将 S 盒的查表改为利用 AES 扩展指令进行运算, 将 S 盒运算变成恒定时间完成.

本文在前人工作的基础上, 综合利用已有的软件优化方法, 提出了一种 ZUC 算法的软件实现的高效组合方法. 我们尝试过的主要方法有: (1) 优化函数的调用过程; (2) 编译器优化技术; (3) 使用延迟取模的方式减少取模次数; (4) 通过合并 S 盒的方式减少访存次数; (5) 利用 SIMD 指令集实现 LFSR 的多轮计算; (6) 将算法分两个线程运行; (7) 使用循环数组的方式优化 LFSR 的移位操作; (8) 使用查表的方式实现了线性变换 L.

通过分析观察发现, 这些方法中, 有些方法只在理论中有效, 但是实际使用的过程中, 并不能减少软件的运行时间; 有些方法单独使用时可以提高效率, 但是当同其它的优化方式共同使用的时候, 其反而会起到反作用. 我们通过尝试组合不同的优化方式, 得到了一种比较高效的软件优化组合方法, 在 Intel Core i7-8750H@2.20 GHz 处理器上, 生成长密钥流时, 软件运行速度可以达到 4.22 Gbps. 和已有的最新结果 3.34 Gbps 相比速度提高 26%. 将该组合方法应用到 ZUC-256 密码算法时, 密钥流产生速度可达到 4.19 Gbps.

综上所述, 本文的主要贡献如下:

(1) 提出了一种高效的软件实现优化组合方式, 使得 ZUC 算法的软件实现速度达到 4.22 Gbps. 和已有的最新结果 3.34 Gbps 相比速度提高 26%.

(2) 本方法可以完全应用于 ZUC-256 算法的软件快速实现. 使用该方法优化 ZUC-256 算法, 密钥流产生速度可以达到 4.19 Gbps.

2 ZUC 算法描述

2.1 ZUC 算法结构

ZUC 算法逻辑上分为三层. 上层是 16 级线性反馈移位寄存器, 包含 16 个 31 比特寄存器变量. 中层为比特重组. 下层为非线性函数 F. ZUC 算法的结构图如图 1 所示.

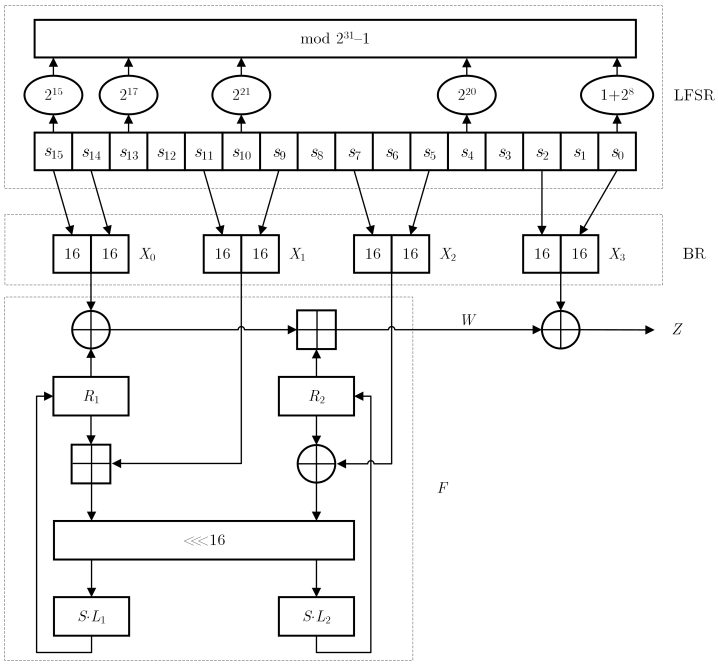

图 1 ZUC 算法结构图

Figure 1 ZUC algorithm structure chart

ZUC 算法以 128 比特的初始密钥 k 和 128 比特的初始向量 iv 作为输入, 在密钥字输出阶段, 每运行一个节拍, 依次调用比特重组、F 函数、LFSR 工作模式, 并产生一个 32 比特的密钥字 Z.

2.2 线性反馈移位寄存器 (LFSR)

LFSR 有 2 种运行模式: 初始化模式和工作模式. LFSR 在初始化模式下接收一个 31 比特字 u, u 是由非线性函数 F 的 32 比特输出 W 通过舍弃最低位比特得到, 即 $u = W \gg 1$. LFSR 的初始化模式计算过程如下:

LFSRWithInitialisationMode(u)

{

$v = 2^{15}s_{15} + 2^{17}s_{13} + 2^{21}s_{10} + 2^{20}s_4 + (1 + 2^8)s_0 \bmod (2^{31} - 1)$;

$s_{16} = (v + u) \bmod (2^{31} - 1)$;

if $s_{16} = 0$; then set $s_{16} = 2^{31} - 1$;

$(s_1, s_2, \cdots, s_{16}) \to (s_0, s_1, \cdots, s_{15})$;

}

LFSR 的工作模式不接收任何输入, 其计算过程如下:

LFSRWithWorkMode()

{

$s_{16} = 2^{15}s_{15} + 2^{17}s_{13} + 2^{21}s_{10} + 2^{20}s_4 + (1 + 2^8)s_0 \bmod (2^{31} - 1)$;

if $s_{16} = 0$; then set $s_{16} = 2^{31} - 1$;

$(s_1, s_2, \cdots, s_{16}) \to (s_0, s_1, \cdots, s_{15})$;

}

2.3 比特重组

比特重组从 LFSR 的寄存器单元中抽取 128 比特组成 4 个 32 比特字 X_0, X_1, X_2, X_3. 比特重组的计算过程如下:

BitReconstruction()
{
$X_0 = s_{15H}||s_{14L}$;
$X_1 = s_{11L}||s_{9H}$;
$X_2 = s_{7L}||s_{5H}$;
$X_3 = s_{2L}||s_{0H}$;
}

其中下标 H 和 L 分别表示高 16 比特和低 16 比特.

2.4 非线性函数 F

F 包含 2 个 32 比特的记忆单元变量 R_1 和 R_2. F 的输入为 3 个 32 比特字 X_0, X_1, X_2, 输出为一个 32 比特字 W, 计算过程如下:

$F(X_0, X_1, X_2)$
{
$W = ((X_0 \oplus R_1) + R_2) \bmod 2^{32}$;
$W_1 = (R_1 + X_1) \bmod 2^{32}$;
$W_2 = R_2 \oplus X_2$;
$R_1 = S(L_1(W_{1L}||W_{2H}))$;
$R_2 = S(L_2(W_{2L}||W_{1H}))$;
}

其中 S 为 32 比特的 S 盒变换, 由 4 个小的 8×8 的 S 盒并置而成, 即 (S_0, S_1, S_2, S_3), 其中 $S_0 = S_2, S_1 = S_3$. L_1 和 L_2 为 32 比特线性变换, 定义如下:

$$L_1(X) = X \oplus (X \lll 2) \oplus (X \lll 10) \oplus (X \lll 18) \oplus (X \lll 24)$$
$$L_2(X) = X \oplus (X \lll 8) \oplus (X \lll 14) \oplus (X \lll 22) \oplus (X \lll 30)$$

其中 $\lll$ 为循环左移.

3 优化方法

3.1 调用优化

3.1.1 使用 inline 函数的方式优化函数调用

在 ZUC 算法的密钥输出阶段, 每运行一个节拍, 需要依次调用比特重组、F 函数、LFSR 工作模式这 3 个函数, 并产生一个 32 比特的密钥字. 频繁的函数调用会大量消耗栈空间. 为了避免函数调用过程中额外的开销, 可以在函数定义前加上 inline, 使其成为内联函数, 函数的代码被放入符号表中, 可以解决频繁调用的函数大量消耗栈空间的问题, 使程序运行速度得到明显提升.

3.1.2 使用宏定义的方式优化函数调用

与使用 inline 一样, 使用宏定义函数可以使函数在程序中直接被替换, 而这一切使用预处理器实现, 没有了参数压栈、代码生成等一系列的操作. 因此实现效率高, 速度提升很明显.

3.1.3 使用寄存器优化程序访存

对于函数中频繁使用的变量, 可以定义为寄存器变量, 即将其存储在寄存器中, 省去了内存与 CPU 的数据交换过程, 使取值速度得到提高. 但是由于寄存器有限, 并不是所有变量都能定义为寄存器变量.

3.2 编译优化

3.2.1 提高编译器自动优化等级的方式优化程序运行

使用 gcc 进行编译, 优化等级选择 O3. 在该等级下, 编译器会对代码的分支、表达式、常量来进行优化, 加入寄存器的使用, 同时还会进行分支预测: 对循环每一层的预测, 以便于将循环拆分, 可以提高执行效率. 编译器还会试图用已有的值来代替未知的值, 并且还会用加代替乘 (因为运算器的特性, 乘法十分复杂耗时). 在该命令下, 程序运行速度得到极大提升.

3.2.2 使用编译器对于 Core i7 处理器的本地运行性能进行优化

使用 gcc 进行编译时加入-march = native 命令, 在编译程序时, 借助参数传递的方法, 使用与系统 CPU 相匹配的 gcc 参数, 编译出的程序就是为系统 CPU 而进行特定优化过的, 因而执行速度和效率都会是最好. 该方法在优化等级 O3 的基础上, 使程序速度进一步提升.

3.2.3 使用编译器展开循环的方式进行优化

程序中存在大量的循环, 因此在使用 gcc 编译时加入-funroll-loops 命令, 对程序中的循环进行展开. 这样做一是减少了对循环没有直接贡献的计算, 比如循环计数变量的计算, 分支跳转指令的执行等; 二是提供了进一步利用机器特性进行的优化的机会.

3.3 延迟取模

LFSR 的工作模式中, 新的 s_{16} 计算方式如下式所示:

$$s_{16} = 2^{15}s_{15} + 2^{17}s_{13} + 2^{21}s_{10} + 2^{20}s_4 + (1+2^8)s_0 \bmod (2^{31}-1)$$

为了防止中间结果的溢出, 上式总共进行了 5 次模乘和 5 次模加. ZUC 算法官方文档给出了模加和模乘的快速实现方式如下.

当其中一个字具有较低的汉明重量时, 两个 31 比特字模 $2^{31}-1$ 可以通过 31 比特的循环移位和模 $2^{31}-1$ 加法运算实现. 例如计算 $ab \bmod (2^{31}-1)$, 其中 $b = 2^i + 2^j + 2^k$, 则有:

$$ab \bmod (2^{31}-1) = (a \lll_{31} i) + (a \lll_{31} j) + (a \lll_{31} k) \bmod (2^{31}-1)$$

两个 31 比特字 a 和 b 模 $2^{31}-1$ 加法运算 $c = a + b \bmod (2^{31}-1)$ 可以通过以下两步计算实现:

$$\begin{aligned} c &= a + b \\ c &= (c \ \& \ \text{0x7FFFFFFF}) + (c \gg 31) \end{aligned}$$

Avanzi 等人 [6] 提出将 LFSR 中的 5 次模加和 5 次模乘改为普通的加法和乘法, 最后再对结果进行两次模运算, 同时为了防止中间结果的溢出, 使用 64 比特的整型变量存储中间结果. 计算过程如下所示:

$$\begin{aligned} t &= 2^{15}s_{15} + 2^{17}s_{13} + 2^{21}s_{10} + 2^{20}s_4 + (1+2^8)s_0 \\ t &= (t \ \& \ \text{0x7FFFFFFF}) + (t \gg 31) \\ s_{16} &= (t \ \& \ \text{0x7FFFFFFF}) + (t \gg 31) \end{aligned}$$

通过延迟取模, LFSR 每轮的模运算次数由 10 次降低到了 2 次.

3.4 S 盒合并

S 盒变换的结构如图 2 所示, 可以看出, S 盒变换输入 32 比特输出 32 比特, 其中又由 4 个 8 比特输入 8 比特输出的小 S 盒组成, 即 $S_0S_1S_0S_1$, 小 S 盒的大小为 0.5 KB. 对于 S 盒变换的任意 32 比特输入 $A = A_0A_1A_2A_3$(A_0, A_1, A_2, A_3 均为 8 比特), 有:

$$S(A) = (S_0(A_0) \ll 24) \oplus (S_1(A_1) \ll 16) \oplus (S_2(A_2) \ll 8) \oplus S_3(A_3)$$

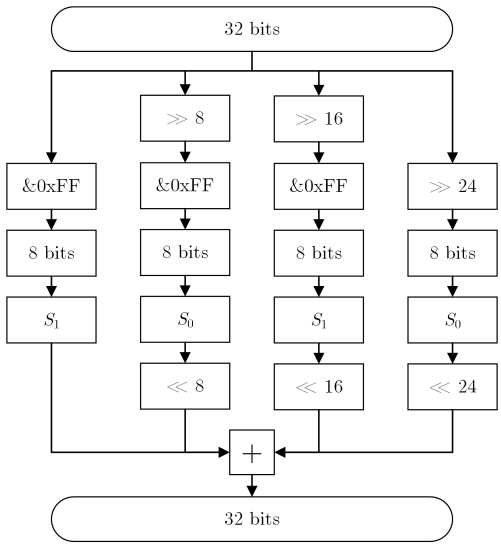

图 2 S 盒变换
Figure 2 S-box substitution

由于小 S 盒排布的对称性, Yu 等人 [8] 提出将 S_0 和 S_1 两个小 S 盒合并成一个 16 比特输入 16 比特输出的大 S 盒, 大 S 盒的大小为 128 KB. 小 S 盒合并后, 对于 S 盒变换的任意 32 比特输入 $A = A_0A_1A_2A_3$ (A_0, A_1, A_2, A_3 均为 8 比特), 有:

$$S(A) = (\text{bigS}(A_0A_1) \lll 16) \oplus \text{bigS}(A_2A_3)$$

合并后的 S 盒变换结构如图3所示.

如表1所示, 将 S_0, S_1 两个小 S 盒合并后, S 盒的存储空间变大了, 但查表的次数和位运算的次数都减少了.

表 **1** S 盒合并前后对比
Table 1 Comparison of S-box before and after combination

	移位 (次)	与 (次)	或 (次)	查表 (次)	S 盒大小 (KB)
合并前	6	3	3	4	0.5
合并后	2	1	1	2	128

更进一步, 将输入的高 16 比特查表后的移位操作合并到大 S 盒中 (即将大 S 盒中的每个元素左移 16 位), 低 16 比特的大 S 盒不变, 形成两个大 S 盒, 分别记为 bigSHigh 和 bigSLow. 合并移位操作后, 每次 S 盒变换都减少了一次移位操作, 但 S 盒的存储空间变大了. 对任意 32 比特输入 $A = A_0A_1A_2A_3$ (A_0, A_1, A_2, A_3 均为 8 比特), 有:

$$S(A) = \text{bigSHigh}(A_0A_1) \oplus \text{bigSLow}(A_2A_3)$$

高位 S 盒合并移位后的 S 盒变换结构图如图4所示.

实验结果: 在测试中, 合并 S 盒后速度加快, 但是使用两个大 S 盒速度反而有所下降, 这可能是由于两个大 S 盒占用空间过大, 导致缓存命中率降低, 最终我们使用的是将 S 盒合并为一个大 S 盒.

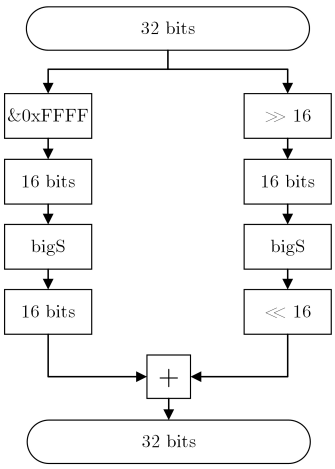

图 3 合并后的 S 盒变换
Figure 3 S-box substitution after merging

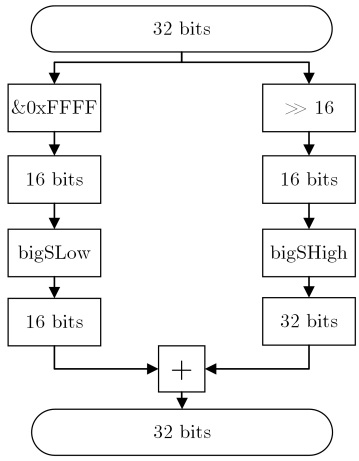

图 4 合并移位操作的 S 盒变换
Figure 4 S-box substitution of merging shift operation

3.5 使用 SIMD 指令集

3.5.1 SIMD 指令集简介

SIMD(单指令多数据) 指令集支持向量化的数据并行, 一个指令可以同时操作多个数据. 本文使用的 AVX 及 AVX2 指令集最多可支持 256 比特宽的向量. SIMD 指令集主要包括与、或、异或、左移、右移等逻辑运算, 加、减、乘、除、绝对值等算数运算以及比较、加载或存储等. 本文用到的 SIMD 指令如表 2 所示.

表 **2** SIMD 指令
Table 2 SIMD instruction

C/C++ 接口	功能描述
_mm256_set1_epi64x	4 道 64 位全部设置为某值
_mm256_slli_epi64	4 道 64 位左移
_mm256_add_epi64	4 道 64 位加法
_mm256_maskload_epi64	从内存中加载 4 道 64 位整数
_mm256_maskstore_epi64	将 4 道 64 位整数存储到内存中

3.5.2 在线性反馈移位寄存器中使用 SIMD 指令集

考虑在 LFSR 中一次性生成 4 个新的 s_i, 表达式如下所示:

$$
\begin{aligned}
s_{16} &= (2^{15}s_{15} + 2^{17}s_{13} + 2^{21}s_{10} + 2^{20}s_4 + 2^8 s_0 + s_0) \bmod (2^{31}-1)\\
s_{17} &= (2^{15}s_{16} + 2^{17}s_{14} + 2^{21}s_{11} + 2^{20}s_5 + 2^8 s_1 + s_1) \bmod (2^{31}-1)\\
s_{18} &= (2^{15}s_{17} + 2^{17}s_{15} + 2^{21}s_{12} + 2^{20}s_6 + 2^8 s_2 + s_2) \bmod (2^{31}-1)\\
s_{19} &= (2^{15}s_{18} + 2^{17}s_{16} + 2^{21}s_{13} + 2^{20}s_7 + 2^8 s_3 + s_3) \bmod (2^{31}-1)
\end{aligned}
$$

在生成新的 s_i 前, $s_0, s_1, \cdots, s_{15}$ 均已知, 而 $s_{16}, s_{17}, s_{18}, s_{19}$ 每一项的计算均依赖前一项计算的结果. 因此我们将最后五项的计算进行并行处理, 在处理完并行部分后, 再依次处理不可并行的部分, 依次生成 $s_{16}, s_{17}, s_{18}, s_{19}$.

由于使用了延迟取模的方法, 中间变量需要用 64 位的变量存储, 因此使用 256 位宽的寄存器去存储

四个表达式中操作相同的数据, 如下所示:

$$\begin{aligned} r_0 &= \{s_0, s_1, s_2, s_3\};\\ r_1 &= \{s_4, s_5, s_6, s_7\};\\ r_2 &= \{s_{10}, s_{11}, s_{12}, s_{13}\};\\ r_3 &= \{s_{13}, s_{14}, s_{15}, s_{16}\} \end{aligned}$$

其中, s_{16} 尚未计算, 先设为 0. 之后我们就可以使用一个指令对多个数据同时进行加法和移位, 减少操作次数. 使用 SIMD 指令集的并行部分如下所示:

$$\begin{aligned} r_0 &= r_0 + (r_0 \ll 8)\\ r_1 &= r_1 \ll 20\\ r_2 &= r_2 \ll 21\\ r_3 &= r_3 \ll 17\\ r_4 &= r_0 + r_1 + r_2 + r_3\\ \{s_{16}, s_{17}, s_{18}, s_{19}\} &= r_4 \end{aligned}$$

最后处理必须串行的部分, 依次计算 $s_{16}, s_{17}, s_{18}, s_{19}$, 如下所示:

$$\begin{aligned} s_{16} &= (2^{15}s_{15} + s_{16}) \bmod (2^{31} - 1)\\ s_{17} &= (2^{15}s_{16} + s_{17}) \bmod (2^{31} - 1)\\ s_{18} &= (2^{15}s_{17} + s_{18}) \bmod (2^{31} - 1)\\ s_{19} &= (2^{17}s_{16} + 2^{15}s_{18} + s_{19}) \bmod (2^{31} - 1) \end{aligned}$$

3.6 软件并行方法

使用多线程的方法尝试并行处理数据. 考虑到 LFSR 和 BR 与 F 函数的独立性, 建立两个线程: 一个线程运行 LFSR, 另一个线程运行 BR 和 F 函数. 通过共享变量 ready 保证两个线程的同步; 使用高性能的自旋锁, 通过线程的忙式等待保证对 ready 访问的互斥性.

但是, 由于线程运行的每一轮时间过于短暂, 自旋锁的获取和释放时间实际上占了很大一部分, 导致速度反而有了明显的下降. 因此, 在单组数据内部软件多线程并行的方法不可用.

3.7 使用循环数组的方式优化 LFSR 的循环移位

在 LFSR 部分, 每一轮都是以 16 次赋值结束的. 通过观察我们发现, 每一轮进行的都是移位操作, 真正被 “改变” 的只有一个 31 比特数. 因此, 我们用循环数组实现移位寄存器, 使用头来寻址, 将每轮的 16 次赋值改为了对队头的赋值以及队头的后移, 有效地减少了赋值操作, 提高了程序运行效率.

3.8 使用查表的方式实现 L 变换

对于线性变换 L 的任意 32 比特输入 A, 记为 $A = A_0A_1A_2A_3$, 其中 A_i 为 8 比特, 由于变换 L 的线性性质, 有:

$$L_i(A) = L_i(A_0 \ll 24) \oplus L_i(A_1 \ll 16) \oplus L_i(A_2 \ll 8) \oplus L_i(A_3)$$

因此, 对于每个线性变换 L_i, 可以提前计算好 $L_i(A_0 \ll 24), L_i(A_1 \ll 16), L_i(A_2 \ll 8), L_i(A_3)$ 的值, 生成四个 8 比特输入 32 比特输出的表 $L_i\mathrm{Box}_j$, 每张表的大小为 256×32 bits. 这样, 线性变换 L_i 的实现就由位运算变为了四次查表结果的异或, 即:

$$L_i(A) = L_i\mathrm{Box}_0(A_0) \oplus L_i\mathrm{Box}_1(A_1) \oplus L_i\mathrm{Box}_2(A_2) \oplus L_i\mathrm{Box}_3(A_3)$$

实验结果: 在实际实现过程中发现, L 变换的两种实现方法速度提升不大, 可能是因为四次寻址, 访存用时较多, 但是改用两个 16 bits 数查表又会导致单次寻址时间增加, 导致速度下降.

4 实验及结果

4.1 实验环境和测试数据的选择

使用 Dell g3 3579 笔记本作为测试环境.

CPU: Intel Core i7-8750H.

内存: 16 G.

编译器: 开源的 GNU 编译器 gcc.

系统测试方案: 对于每一种对方案的改进, 测试方案分为两步, 首先, 使用算法标准中的测试样例对于算法的正确性进行验证, 然后, 生成大量的密钥流对于算法的性能进行验证. 4.2 节在验证了算法正确性的基础上, 对算法的性能进行了测试.

测试使用的数据量: 每次生成 128 比特密钥流, 循环生成 20 M (20 971 520) 轮作为一次测试.

4.2 单个优化方法

4.2.1 基准测试

对不做任何优化的 ZUC 算法进行性能测试, 测试结果如表 3 所示. 在基准测试中, 程序平均用时 3491 ms. 平均运行速率为 733 Mbps.

表 **3** 基准测试

Table 3 Benchmark

测试序号	运行时间 (ms)	测试序号	运行时间 (ms)
1	3515	6	3475
2	3476	7	3514
3	3472	8	3499
4	3468	9	3521
5	3471	10	3503

4.2.2 优化函数调用过程

优化 ZUC 算法的函数调用过程, 并对性能进行测试, 测试结果如表 4 所示. 在优化函数的调用过程后, 程序平均用时 3013 ms, 加密速度能够达到 850 Mbps.

表 **4** 优化函数的调用过程

Table 4 Optimizing calling process of function

测试序号	运行时间 (ms)	测试序号	运行时间 (ms)
1	3018	6	3018
2	2978	7	3007
3	2997	8	3012
4	3047	9	3003
5	3024	10	3024

4.2.3 编译优化

对 ZUC 算法进行编译优化, 并测试性能, 结果如表 5 所示, 程序平均运行时间为 1231 ms, 平均加密速度为 2079 Mbps.

4.2.4 延迟取模

使用延迟取模的方法对 ZUC 算法的 LFSR 进行优化, 并测试性能, 结果如表 6 所示. 使用延迟取模之后, 程序平均运行时间为 2371 ms, 平均加密速度为 1080 Mbps.

表 **5** 编译优化
Table 5 Compile optimization

测试序号	运行时间 (ms)	测试序号	运行时间 (ms)
1	1238	6	1239
2	1215	7	1230
3	1228	8	1239
4	1233	9	1230
5	1229	10	1231

表 **6** 延迟取模
Table 6 Modulo delay

测试序号	运行时间 (ms)	测试序号	运行时间 (ms)
1	2387	6	2376
2	2365	7	2371
3	2376	8	2352
4	2370	9	2375
5	2376	10	2357

4.2.5 S 盒合并

将非线性函数 F 内的两个 S 盒合并, 并测试性能, 结果如表 7 和表 8 所示.

表 **7** S 盒合并 — 1 个大 S 盒
Table 7 S-box merge — 1 large S-box

测试序号	运行时间 (ms)	测试序号	运行时间 (ms)
1	3357	6	3314
2	3280	7	3292
3	3253	8	3255
4	3286	9	3282
5	3254	10	3254

表 **8** S 盒合并 — 2 个大 S 盒
Table 8 S-box merge — 2 large S-boxes

测试序号	运行时间 (ms)	测试序号	运行时间 (ms)
1	3394	6	3399
2	3369	7	3404
3	3435	8	3387
4	3456	9	3393
5	3395	10	3370

将非线性函数 F 中的两个 S 盒整合成一个大 S 盒之后, 程序平均运行时间为 3282 ms. 平均运行速度为 780 Mbps. 进一步地, 将大 S 盒进行移位操作, 形成 2 个大 S 盒之后, 程序平均运行时间 3400 ms, 平均运行速度 753 Mbps, 可以发现, 使用一个大 S 盒的优化效果更好.

4.2.6 使用 SIMD 指令集

在 LFSR 中使用 SIMD 指令集, 并测试性能, 结果如表 9 所示. 使用 SIMD 的方式优化 LFSR 后, 程序平均运行时间为 2732 ms, 平均运行速度为 937 Mbps.

4.2.7 软件并行方法

使用基于多线程的优化方案对 ZUC 进行软件并行, 并测试性能, 结果如表 10 所示. 在使用了线程同步之后, 程序的平均运行时间达到 19 959 ms, 相比于基准测试, 速度大幅下降.

表 **9** 使用 SIMD 指令集
Table 9 Using SIMD instruction

测试序号	运行时间 (ms)	测试序号	运行时间 (ms)
1	2739	6	2737
2	2743	7	2715
3	2717	8	2741
4	2745	9	2714
5	2732	10	2742

表 **10** 软件并行
Table 10 Software parallelism

测试序号	运行时间 (ms)	测试序号	运行时间 (ms)
1	19 811	6	21 542
2	19 738	7	20 706
3	19 145	8	19 812
4	19 865	9	19 671
5	19 746	10	19 558

4.2.8 使用循环数组实现 LFSR 的循环移位

使用循环数组实现 LFSR 的循环移位, 并测试性能, 结果如表 11 所示. 使用循环数组的方式来优化 LFSR 的循环移位后, 程序平均运行时间为 3051 ms, 平均运行速度为 839 Mbps.

表 **11** 循环数组实现 LFSR 的循环移位
Table 11 Using cyclic array to realize cyclic shift of LFSR

测试序号	运行时间 (ms)	测试序号	运行时间 (ms)
1	3089	6	3052
2	3033	7	3044
3	3047	8	3046
4	3056	9	3054
5	3063	10	3028

4.2.9 使用查表的方式实现两个 L 变换

使用查表的方式实现非线性函数 F 中的两个 L 变换, 并测试性能, 结果如表 12 所示. 将两个线性变换用查表代替之后, 平均运行时间为 3620 ms, 平均运行速度为 707 Mbps. 相对于基准测试速度略有下降.

表 **12** 查表实现 L 变换
Table 12 Using look up table to realize L transformation

测试序号	运行时间 (ms)	测试序号	运行时间 (ms)
1	3683	6	3593
2	3652	7	3596
3	3592	8	3613
4	3622	9	3590
5	3641	10	3625

4.2.10 总结

本节尝试使用各种基本的优化方式对 ZUC 算法进行优化, 其中被证明有效的优化方式为优化函数的调用过程, 使用编译器进行优化, 合并两个 S 盒, 使用 SIMD 指令集对于 LFSR 进行优化, 使用循环数组方式的 LFSR. 被证明无效的方式为使用多线程的方式优化 ZUC 算法, 使用查表的方式优化线性变换 L.

4.3 组合性能测试

组合性能测试中, 我们将不同的优化方式进行组合, 以达到更高的速度. 在下面的组合速度测试中, 使用 “调用” 表示优化函数的调用过程, 使用 “编译” 表示编译器优化, 使用 “循环数组” 表示利用循环数组实现 LFSR 的循环移位, 使用 “S” 表示将两个 S 盒合并为一个大 S 盒, 使用 “SIMD” 表示使用 SIMD 指令集优化 LFSR. 测试的结果如表 13 所示. 各种优化方式的有效性如表 14 所示.

表 **13** 组合性能测试

Table 13 Combined performance test

测试序号	使用的优化方式	平均运行时间 (ms)	平均运行速度 (Gbps)	提升百分比 (%)
1	基准测试	3491	0.716	0
2	调用	3013	0.830	16
3	调用 + 编译	955	2.62	266
4	调用 + 编译 + 循环数组	973	2.57	259
5	调用 + 编译 + 延迟取模	782	3.20	347
6	调用 + 编译 + 延迟取模 + S	593	4.22	489
7	调用 + 编译 + 延迟取模 + S + SIMD	682	3.66	412
8	调用 + 编译 + 延迟取模 + SIMD	704	3.55	396

表 **14** 各种优化方式的有效性

Table 14 Effectiveness of various optimization methods

优化方式	单独使用的有效性	与其他优化方式结合的有效性
优化函数调用和变量存取	✓	✓
编译优化	✓	✓
延迟取模	✓	✓
S 盒合并 (1 个大 S 盒)	✓	✓
S 盒合并 (2 个大 S 盒)	×	×
SIMD 指令	✓	×
软件并行	×	×
循环数组优化 LFSR	✓	×
查表实现 L 变换	×	×

在将优化方式进行组合的过程中, 在使用了优化函数的调用过程和编译器优化之后, 我们发现再使用循环数组的方式对于 LFSR 进行优化的话, 反而会拖慢程序的运行效率, 因此在后面的组合优化过程中, 不再使用循环数组的方式对于 LFSR 进行优化, 在使用了优化函数的调用过程, 编译优化, 延迟取模之后, 对于合并 S 盒和 SIMD 优化的处理过程中, 我们发现使用这两种方式都可以使速度得到提高, 但是同时使用这两种方法的优化效果反而不如仅仅使用合并 S 盒的优化方式. 最终, 我们的实验结果是使用优化函数的调用过程、编译器优化、延迟取模、合并 S 盒可以获得最快的密钥流生成速度.

4.4 加密数据测试

加密数据测试的结果如表 15 所示, 在加密数据测试中, 每初始化一次密钥加密 64 KB 数据获得了最快的速度, 最快的速度是 0.849 bits/cycle. 数据较短时, 随着单次加密数据的变长. 速度提升的原因是初始化在运行中所占的时间变少. 但是观察到随着数据的加长, 速度有些下降, 这可能是存储数据的内存过长导致缓存命中率下降导致的. 在密钥流生成测试中没有出现类似的现象是因为密钥流生成的过程中, 循环覆盖存储密钥流的内存, 没有使用很长的内存区域.

4.5 ZUC-256 应用测试

考虑到 ZUC-256 算法与 ZUC-128 算法在结构上具有高度的相似性, 我们使用的优化方法可以方便地应用于 ZUC-256 算法的软件优化工作. 我们将经过测试效果较好的优化函数调用和变量存取、编译优

表 **15** 加密数据测试
Table 15 Data encryption test

数据长度 (B)	未优化 (bits/cycle)	优化后 (bits/cycle)	提升 (%)
128	0.079	0.440	454
1K	0.148	0.827	461
8K	0.166	0.842	408
64K	0.168	0.849	406
512K	0.168	0.840	400
4M	0.164	0.819	400
32M	0.163	0.802	393
320M	0.153	0.522	242

化、延迟取模和两个 S 盒合并应用到 ZUC-256 算法的实现中, 进行了推广测试, 测试的平均运行时间 596 ms, 平均运行速度 4.19 Gbps.

5 总结

本文对于 ZUC 算法的软件实现尝试了 8 种优化方法, 分别是优化函数的调用过程、编译器优化、通过延迟取模的方法优化 LFSR、通过合并两个 S 盒的方式来减少访存的次数、利用 SIMD 指令集实现 LFSR 的多轮计算、将算法分为两个线程执行、使用循环数组的方式优化 LFSR、使用查表的方式实现线性变换 L. 其中, 将算法分为两个线程进行执行的方式, 由于线程的同步开销过大, 速度明显下降, 使用查表的方式实现线性变换 L 也会使速度下降. 除此之外, 其他的方法单独使用时是有效的. 然后, 我们将各种优化方式进行整合, 在最终实现中, 我们使用了效果较好的优化函数调用和变量存取、编译优化、延迟取模和合并两个 S 盒, 最终使用 ZUC 算法生成长密钥流的实现速度可以达到 4.22 Gbps. 将该方法应用到 ZUC-256 算法中, 生成密钥流速度可以达到 4.19 Gbps. 实验源代码见 https://github.com/zzz136454872/ZUC_software_optimization.

参考文献

[1] 3GPP Approves ZUC Cryptographic Algorithm as an International Standard[J]. World Telecommunications, 2011, (10): 9.
3GPP 批准我国祖冲之密码算法成为国际标准 [J]. 世界电信, 2011, (10): 9.

[2] Internationalization of Domestic Cryptographic Algorithm Standards Creates New Achievements ZUC Algorithm Officially Becomes the ISO/IEC International Standard[EB/OL]. http://www.oscca.gov.cn/sca/xwdt/2020-05/11/content_1060747.shtml. 2020.
国产密码算法标准国际化再创佳绩 ZUC 算法正式成为 ISO/IEC 国际标准 [EB/OL]. http://www.oscca.gov.cn/sca/xwdt/2020-05/11/content_1060747.shtml. 2020.

[3] Design Team. ZUC-256 stream cipher[J]. Journal of Cryptologic Research, 2018, 5(2): 167–179. [DOI: 10.13868/j.cnki.jcr.000228]
ZUC 算法研制组. ZUC-256 流密码算法 [J]. 密码学报, 2018, 5(2): 167–179. [DOI: 10.13868/j.cnki.jcr.000228]

[4] ZHANG Q L, LIU Z B, LI M, et al. A high-throughput unrolled ZUC core for 100Gbps data transmission[C]. In: Information Security and Privacy—ACISP 2014. Springer Cham, 2014: 370–385. [DOI: 10.1007/978-3-319-08344-5_24]

[5] GB/T 33133.1-2016. Information Security Technology—ZUC Stream Cipher Algorithm—Part 1: Algorithm Description[S]. 2016.
GB/T 33133.1-2016. 信息安全技术 祖冲之序列密码算法第 1 部分: 算法描述 [S]. 2016.

[6] AVANZI R M, BRUMLEY B B. Faster 128-EEA3 and 128-EIA3 software[C]. In: Information Security—ISC 2013. Springer Cham, 2015: 199–208. [DOI: 10.1007/978-3-319-08344-5_24]

[7] LI J P, ZHANG Y S, DONG X Z, et al. Fast software implementation of ZUC-256 algorithm[J]. Application Research of Computers, 2019, 36(12): 3797–3800. [DOI: 10.19734/j.issn.1001-3695.2018.07.0581]
李建鹏, 张艳硕, 董秀则, 等. ZUC-256 算法的快速软件实现 [J]. 计算机应用研究, 2019, 36(12): 3797–3800. [DOI: 10.19734/j.issn.1001-3695.2018.07.0581]

[8] YU K, GU N J, SU J J, et al. Efficient software implementation of ZUC stream cipher[C]. In: Proceedings of the 2nd International Conference on Vision, Image and Signal Processing. ACM, 2018: 1–6. [DOI: 10.1145/3271553.3271580]

[9] DRUCKER N, GUERON S. Fast constant time implementations of ZUC-256 on x86 CPUs[C]. In: Proceedings of 2019 16th IEEE Annual Consumer Communications & Networking Conference (CCNC). IEEE, 2019: 1–7. [DOI: 10.1109/CCNC.2019.8651851]

作者信息

张宇鹏 (1998–), 河北石家庄人, 本科生. 主要研究领域为密码算法的设计与实现.
yupengzhang@buaa.edu.cn

高莹 (1977–), 湖北大悟人, 博士, 副教授. 主要研究领域为密码学及其应用.
gaoying@buaa.edu.cn

严宇 (1999–), 安徽六安人, 本科生. 主要研究领域为密码算法的设计与实现.
yanyu0324@buaa.edu.cn

刘翔 (2000–), 江西赣州人, 本科生. 主要研究领域为密码算法的设计与实现.
lx1234@buaa.edu.cn