密码学报 ISSN 2095-7025 CN 10-1195/TN
Journal of Cryptologic Research, 2021, 8(3): 521–536

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

ZUC-256 算法的快速软件实现

白 亮, 贾文义, 朱桂桢

兴唐通信科技有限公司, 北京 100191
通信作者: 白亮, E-mail: gmu.shmily@gmail.com

摘 要: 序列密码算法 ZUC-128 是 3GPP 机密性算法 EEA3 和完整性算法 EIA3 的核心. ZUC-256 算法是 ZUC-128 的升级版本, 以应对 5G 通信安全性需要和后量子技术安全挑战. 本文探讨 ZUC-256 流密码算法在 x86 架构处理器上的软件优化实现方法. 我们利用单指令多数据 (Single Instruction Multiple Data, SIMD) 技术在 ZUC-256 密钥流算法已有的快速软件实现的基础上, 作更进一步的优化, 并给出消息认证码生成算法的软件优化实现. 在 Intel Xeon Gold 6128 处理器上, 优化后的密钥流算法在 16 个密钥或 16 个 IV 条件下的软件实现性能可以达到 21 Gbps, 超过了 5G 通信加密标准中的下行速度要求. 通过实验对比, 密钥流算法的实现性能在已有的结果上最多提升了 56%; 同无优化的消息认证码实现方法相比, 我们利用 SIMD 技术的软件实现性能提高了 20 倍.

关键词: ZUC-256; 序列密码; 软件优化; SIMD
中图分类号: TP309.7 **文献标识码:** A DOI: 10.13868/j.cnki.jcr.000455

中文引用格式: 白亮, 贾文义, 朱桂桢. ZUC-256 算法的快速软件实现[J]. 密码学报, 2021, 8(3): 521–536. [DOI: 10.13868/j.cnki.jcr.000455]

英文引用格式: BAI L, JIA W Y, ZHU G Z. Efficient software implementations of ZUC-256[J]. Journal of Cryptologic Research, 2021, 8(3): 521–536. [DOI: 10.13868/j.cnki.jcr.000455]

Efficient Software Implementations of ZUC-256

BAI Liang, JIA Wen-Yi, ZHU Gui-Zhen

Xingtang Communication Technology Co. Ltd., Beijing 100191, China
Corresponding author: BAI Liang, E-mail: gmu.shmily@gmail.com

Abstract: ZUC algorithm design team proposed the ZUC-256 stream cipher algorithm to meet the requirements in the upcoming 5G technology and post quantum cryptography. The ZUC-256 algorithm is an upgraded version of ZUC-128, and the latter is the core of EEA3 encryption algorithm and EIA3 integrity algorithm in 3GPP. To apply ZUC-256 widely on various platforms and environments, having efficient and stable implementations on both software and hardware is a precondition, which means that the fast software implementation of ZUC-256 is of great importance. This paper explores the software implementation optimizations of ZUC-256 stream cipher for x86 processors. The software implementation of ZUC-256 encryption algorithm is optimized using Single Instruction Multiple Data (SIMD), and the software implementation optimization of MAC algorithm is proposed. Using the proposed optimization techniques, this paper obtains a software performance speed up to 21 Gbps with 16 keys or 16 initial values input for Intel Xeon Gold 6128 processor, which exceeds the downlink speed requirement in the 5G communication standard. Experiments show that some of the results reach

收稿日期: 2020-12-31 定稿日期: 2021-02-05

up to 56% improvement compared with the existing results. And using SIMD technique, a speedup of up to 20x can be obtained compared with the original implementation without optimization on MAC.

Key words: ZUC-256; stream ciphers; software optimizations; SIMD

1 引言

祖冲之序列密码算法 (即 ZUC 算法, 为了和 ZUC-256 算法区分, 下文中我们将其称为 ZUC-128 算法) 使用 128 比特密钥, 能够提供数据的机密性和完整性保护功能. 2011 年 9 月在日本福冈召开的第三代合作伙伴计划 (3GPP) [1] 系统架构组 (SA) 会议上, ZUC-128 算法被批准写入 3GPP 长期演进 (LTE) 系统标准规范 [2], 用于 LTE 空口数据传输保护, 是我国首个成为国际标准的密码算法. 3GPP 基于 ZUC-128 算法, 结合 LTE 系统实际工作输入输出需要, 制定了机密性保护算法 (128-EEA3) 使用规范和完整性保护算法 (128-EIA3) 使用规范. 在 5G R15 版本标准制定过程中, 3GPP 中国参会企业代表与 ETSI SAGE、GSMA 等共同对 128-EEA3/EIA3 标准进行了修订, 将 ZUC-128 算法作为 5G 空口机密性和完整性保护算法 (128-NEA3/NIA3) 推进成为了 5G 标准. 随着量子计算不断发展, 在 2017 年 3GPP 安全组 (SA3) 的会议上, 多个企业提出了 5G 中使用 256 比特对称密码算法的需求. 为此, 3GPP SA3 专门开展研究项目, 探讨 256 比特密码算法在 5G 中应用相关问题 [3]. 在研究项目中, 空口机密性和完整性保护算法是各国企业关注的重点: 美国 AT&T 等公司提出未来 5G 标准应当使用 AES-256 算法, 提供抗量子计算攻击的能力; 我国于 2018 年初公布了使用 256 比特密钥、基于 ZUC-128 算法设计的 ZUC-256 序列密码算法 (草案) [4]; 欧洲爱立信等公司在 2018 年底提出了 256 比特的 SNOW-V 序列密码算法 [5]. 与 128 比特密钥的 ZUC-128 算法相比, ZUC-256 算法对初始化进行了重新设计, 增加了初始化向量 (IV) 长度以提供足够的安全冗余; 能够支持多种长度的消息认证码 (MAC), 以应对 5G 潜在的多安全级别的需求. 除了安全性, 空口密码算法在终端芯片和网络侧设备中实现性能、对于现有设备的影响 (如升级、改造成本影响), 也是 3GPP 重要考察指标. 由于 AES-256 算法与现阶段标准化的 AES-128 算法 (128-NEA2/NIA2) 相比仅增加了轮数, 因此对于终端芯片设计的影响非常小; 在网络侧无论是使用专用硬件实现还是基于通用服务器虚拟化实现 (如使用 Intel X86 架构 CPU 的通用服务器, 可以利用 Intel 专门的指令集进行加速), 性能都能够满足 5G 需求. 欧洲爱立信公司提出的 SNOW-V 算法复用了 AES 轮函数的结构, 在网络侧进行通用服务器虚拟化实现时, 能够直接使用 AES 算法的全部指令集, 加解密速度高于 AES; 但是在终端芯片中很难复用已有硬件. ZUC-256 算法主体结构与 ZUC-128 相同, 对于终端十分友好; 但是 Intel 处理器没有专门针对 ZUC-128 算法设计指令集, 因而 ZUC-256 算法在通用服务器上性能能否满足 5G 通信加密标准的高速要求, 即在纯软件实现条件下达到 20 Gbps 的下行速度 [6], 成为了标准推进中的关键问题.

截至目前, 国内外学者对 ZUC-128 算法实现的研究主要集中在其硬件 FPGA 实现 [7–11] 以及在高通 Hexagon DSP 架构上的软件优化 [12–14]. 然而, 公开的文献中 ZUC-128 的软件实现方法仅限于查表实现, 受限于密钥流生成阶段的查表运算, ZUC-128 算法在软件上的性能表现并不太理想. ZUC-256 算法作为 ZUC-128 算法的升级版本, 也面临着相同的困境. 2008 年 3 月, Intel、AMD 处理器厂商宣布在其 x86 架构处理器上推出用于 AES [15,16] 加/解密运算的 AES-NI 指令集 [17], 采用 AES-NI 指令集的单指令多数据 (Single Instruction Multiple Data, SIMD) 技术 [18,19] 实现 AES 算法的软件实现性能是采用查表方法的 3.5 倍. 近些年来, 随着 SIMD 技术的不断完善, 它的高效软件实现性能使得越来越多的密码算法将其应用到软件实现上, 例如 2012 年, Seiichi 和 Shiho [20] 将 SSE 指令结合 bit-slice 技术 [21] 应用到 PRESENT [22] 和 Piccolo [23] 算法上, 使二者实现吞吐量分别达到 4.73 和 4.57 cycle/byte; 2013 年 Neves 和 Aumasson [24] 将 AVX2 指令集应用到杂凑密码算法 SHA-3 的候选算法 BLAKE [25] 上并提高了其实现性能. 因此, 利用 SIMD 技术快速软件实现 ZUC-256 算法成为重要研究内容. 不久前, 以色列的两位学者利用 AES-NI 指令集对 ZUC-256 的密钥流生成算法的查表运算进行软件优化, 速度提升了 4.5 倍; 其核心思想是直接从 ZUC-256 的 S-盒的数学逻辑结构入手, 利用查表指令和 AES-NI 指令集实现多路并行计算, 从而完全避免建大表 [26]. 但是, 文献 [26] 中仅给出了 ZUC-256 的密钥流生成算法的快速软件优化实现方法, 并没有给出 MAC 生成算法的软件优化实现方法.

本文主要的工作是对 ZUC-256 的软件实现进行整体优化, 包含密钥流生成算法和 MAC 生成算法. 对于密钥流生成算法, 我们对文献 [26] 中的密钥流生成算法实现方法进行优化, 包括 S-盒查表方法以及向量转换的优化等; 对于 MAC 生成算法, 我们将文献 [12] 中对 ZUC-128 的软件优化技术, 包括延迟模约方法和构造泛哈希函数方法, 推广到了 ZUC-256 的 MAC 生成算法中. MAC 生成算法的软件实现主要有两大瓶颈需要突破, 一是明文消息的比特判断, 二是密钥流每次只能进行 "1-比特" 运算. 对于前者我们利用 SIMD 技术中的比较指令来规避比特判断, 据我们所知, 我们是第一个将这种想法应用到序列密码之中的, 但是这种方法依旧无法实现每次按字操作计算. 从 MAC 生成算法的设计角度来看, 密钥流和明文消息可以看成是一个 $\mathbb{GF}(2^m)$ 域上的线性泛哈希函数 [27], 而这刚好是 MAC 生成算法的核心. 2010 年, 英特尔推出的一个新的指令, 无进位乘法指令 PCLMULQDQ, 它最大的特点是支持 $\mathbb{GF}(2^m)$ 域上的乘法, 该指令常常被用于信号处理、有限域、纠错码和密码学中. 对于密码学, 以前的一些研究结果表明可以利用无进位乘法指令来有效实现许多密码算法, 包括但不限于: AES-GCM 中的 GHASH 算法 [28]、二元域上的椭圆曲线 [29,30] 以及 Koblitz 曲线中的椭圆曲线 [31]. 同传统的 $\mathbb{GF}(2^m)$ 域上的乘法相比, 无进位乘法指令在软件实现上有着非常明显的速度提升. 我们利用 PCLMULQDQ 指令将 MAC 生成中的比特滑动异或转化为 $\mathbb{GF}(2^m)$ 上的多项式乘法运算, 达到了每次按字计算而非比特计算的快速软件实现目标, 速度至少可以提升 2.5 倍 [12].

本文探讨 ZUC-256 流密码算法在 x86 架构处理器上的软件优化实现方法, 将 SIMD 技术的并行性应用到 ZUC-256 的软件优化实现中. 实验结果表明, 与目前基于查表实现密钥流生成的方法相比, 利用 SIMD 技术的软件实现性能具有明显优势; 此外利用无进位乘法指令实现 MAC 生成可以做到每次按字操作.

本文的结构如下: 首先第 2 节概述预备知识, 简要介绍 ZUC-256 算法流程. 接下来第 3 节利用 SIMD 技术实现密钥流生成算法以及实现 MAC 生成算法. 在第 4 节, 我们对第 3 节中的方法进行优化以及将其他一些优化方法应用到 ZUC-256, 并对比测试结果. 最后第 5 节对本文做一个总结.

2 预备知识

本节首先介绍一些符号定义, 然后简要介绍 ZUC-256 算法.

2.1 符号定义

下面介绍后文将要使用到的符号定义.

$\oplus$:	异或运算	$\ll$:	逻辑左移
&:	与运算	$\gg$:	逻辑右移
||:	比特串的连接	$\boxplus$:	模 2^{32} 的整数加法运算
$Z \ggg s$:	比特串 Z 循环右移 s 位	$\otimes$:	内积
$Z \lll s$:	比特串 Z 循环左移 s 位		

下文中的十六进制值均以前缀 0x 表示 (如 0x3B 为十进制的 59).

2.2 ZUC-256 密钥流生成算法

ZUC-256 算法的完整描述在文献 [4] 中, ZUC-256 密钥流生成算法包括两个阶段: 初始化阶段和密钥流生成阶段, 这两个阶段操作相似. 每个阶段都包含三个部分:

(1) LFSR. 496-比特长的线性反馈移位寄存器, 由 16 个 31-比特记忆单元变量 $\{s_{15}, s_{14}, \cdots, s_0\}$ 构成, 这些单元均定义在集合 $\{1, 2, \cdots, 2^{31}-1\}$ 上;

(2) BR. 比特重组层, 从 LFSR 层中抽取一些内部状态, 并拼接成 4 个 32-比特字 $\{X_0, X_1, X_2, X_3\}$;

(3) FSM. 有限状态自动机, 输入由 BR 层决定, 包含 2 个 32-比特字 R_1 与 R_2 作为 FSM 中的记忆单元变量.

令 $p = 2^{31}-1$, $\mathbb{H}_p = \{1, 2, \cdots, p\}$ 为不大于 p 的所有正整数集合. 定义 LFSR 反馈函数 $F_1 : \mathbb{H}_p^5 \to \mathbb{H}_p$ 为以下函数:

$$F_1 : (s_0, s_4, s_{10}, s_{13}, s_{15}) \longmapsto (1+2^8)s_0 + 2^{20}s_4 + 2^{21}s_{10} + 2^{17}s_{13} + 2^{15}s_{15}. \tag{1}$$

令 $F_{\mathrm{L}}, F_{\mathrm{H}} : \mathbb{F}_2{}^{32} \to \mathbb{F}_2{}^{16}$ 分别为从 32-比特字 $x = (x_{31}, x_{30}, \cdots, x_1, x_0)$ 中抽取低 16-比特和高 16-比特, 即: $F_{\mathrm{L}} : x \mapsto (x_{15}, x_{14}, \cdots, x_0)$ 和 $F_{\mathrm{H}} : x \mapsto (x_{31}, x_{30}, \cdots, x_{16})$. 令 $L_1, L_2 : \mathbb{F}_2^{32} \to \mathbb{F}_2^{32}$ 为线性函数. 令 $S : \mathbb{F}_{2^8}^4 \to \mathbb{F}_{2^8}^4$ 为 4 个并置 S-盒的非线性函数. 假设 S 函数的输入 $Y = (Y_3, Y_2, Y_1, Y_0)$ 为 4 个 8-比特字节构成的一个 32-比特字, 则 $S(Y) = S_0(Y_3)||S_1(Y_2)||S_0(Y_1)||S_1(Y_0)$, 其中 S_0 与 S_1 是两个给定的 8 进 8 出的 S-盒; 注意两个 S-盒的构造并不一样, S_0 是采用 Feistel 结构构造的, 具有较低的硬件实现面积; 而 S_1 是基于有限域 $\mathbb{GF}(256)$ 上的逆函数构造的, 与分组密码 AES 的 S-盒类似, 它们之间仿射等价. ZUC-256 密钥流生成阶段的每一轮具体步骤描述如下:

(1) $X_0 := F_{\mathrm{H}}(s_{15} \ll 1)||F_{\mathrm{L}}(s_{14})$, $X_1 := F_{\mathrm{L}}(s_{11})||F_{\mathrm{H}}(s_9 \ll 1)$,
$X_2 := F_{\mathrm{L}}(s_7)||F_{\mathrm{H}}(s_5 \ll 1)$, $X_3 := F_{\mathrm{L}}(s_2)||F_{\mathrm{H}}(s_0 \ll 1)$.

(2) $Z := ((X_0 \oplus R_1) \boxplus R_2) \oplus X_3$, $W_1 := X_1 \boxplus R_1$, $W_2 := X_2 \oplus R_2$.

(3) $R_1 := S(L_1(F_{\mathrm{L}}(W_1)||F_{\mathrm{H}}(W_2)))$, $R_2 := S(L_2(F_{\mathrm{L}}(W_2)||F_{\mathrm{H}}(W_1)))$.

(4) $s_{16} := F_1(s_0, s_4, s_{10}, s_{13}, s_{15})$.

(5) $\{s_0, s_1, \cdots, s_{15}\} := \{s_1, s_2, \cdots, s_{16}\}$.

(6) 返回密钥流字 Z.

在初始化阶段, 密钥载入过程首先将一个 256-比特的种子密钥 SK 和一个 184-比特的初始向量 IV 打入到 LFSR 的记忆单元变量 $\{s_{15}, s_{14}, \cdots, s_0\}$ 中作为其初始状态; 其次, 令非线性函数 F 的两个记忆单元变量 R_1 和 R_2 为 0; 最后运行初始化迭代过程 32 次, 完成密钥载入过程. ZUC-256 密钥流生成算法在完成初始化阶段以后, 依次执行 LFSR 层、BR 层和 FSM 层进行状态更新, 完成一次迭代过程, 在此过程中不输出任何密钥流字, 随后进入到密钥流的输出过程.

2.3 ZUC-256 MAC 生成算法

ZUC-256 流密码的 MAC 是基于 ZUC-256 的密钥流字生成的, 其认证标签 Tag 的长度 t 可取为 32, 64 及 128-比特. 令 $M = m_0\|m_1\|\cdots\|m_{l-1}$ 为一段 l-比特长的明文消息, 下面给出 MAC 生成描述:

(1) 运行 ZUC-256 密钥流生成算法产生一段包含 $L = \lceil \frac{l}{32} \rceil + 2 \cdot \frac{t}{32}$ 个字的密钥流. 第一个密钥流字 Z_0 的比特串为 $z_0\|z_1\|\cdots\|z_{31}$, 其中 z_0 和 z_{31} 分别是 Z_0 的最高位和最低位.

(2) 初始化 $\mathrm{Tag} := (z_0, z_1, \cdots, z_{t-1})$.

(3) 当 $i = 0$ 到 $l - 1$ 时执行:

(3.1) $W_i := (z_{t+i}, z_{t+i+1}, \cdots, z_{2t+i-1})$;

(3.2) 若 $m_i = 1$, 则 $\mathrm{Tag} = \mathrm{Tag} \oplus W_i$.

(4) $W_l := (z_{l+t}, z_{l+t+1}, \cdots, z_{l+2t-1})$.

(5) $\mathrm{Tag} = \mathrm{Tag} \oplus W_l$.

(6) 返回 Tag 作为 MAC 输出.

在 MAC 生成阶段, 为了防止伪造攻击, 对于不同长度的认证标签, 在初始化阶段所采用的常数均不相同, 同时也异于 ZUC-256 密钥流生成算法中采用的常数.

3 基于 x86 平台上的 SIMD 技术并行实现 ZUC-256

SIMD 最大的优点在于它的并行性, 该技术可实现同一操作并行处理多组数据. 目前支持 SIMD 技术的处理器厂商主要有 Intel、AMD、ARM 等, Intel 处理器中的 SSE/AVX 指令集及 AMD 处理器中的 SSE/XOP 指令集中的指令均采用 SIMD 技术, 大多数 PC 及服务器采用的是 Intel 处理器. 本文使用的是基于 Intel 处理器的 SIMD 并行技术.

SIMD 技术可用于并行计算, 寄存器的长度有 128 比特 (SSE 系列)、256 比特 (AVX2) 和 512 比特 (AVX512), 使用 SIMD 技术对 ZUC-256 算法进行优化的点主要是考虑可以并行计算逻辑运算、查表运算和模加运算等. 由于 ZUC-256 中计算的基本单元为 32-比特的字, 因此, 除了建大表, 对单密钥或单 IV 模式 (不使用 SIMD 进行并行计算的算法实现模式) 的优化基本不可行. 以下针对 ZUC-256 的优化均为多密钥或多 IV 模式下的优化.

(1) 对于支持 128-比特寄存器的系统 (SSE), 可以并行处理 4 条数据的运算操作;

(2) 对于支持 256-比特寄存器的系统 (AVX2), 可以并行处理 8 条数据的运算操作;

(3) 对于支持 512-比特寄存器的系统 (AVX512), 可以并行处理 16 条数据的运算操作.

接下来我们通过利用 SIMD 技术分别优化实现 ZUC-256 的密钥流生成算法和 MAC 生成算法.

3.1 密钥流生成算法的 SIMD 实现

从 ZUC-256 密钥流生成算法的描述可知, 在 FSM 层需要用到两个 8-比特的 S-盒对 32-比特的字进行四次查表运算. 在单密钥模式中, 我们可以通过建大表以空间换时间的方式快速实现查表运算, 但如果在多密钥模式中使用 SIMD 技术并行处理多条数据, 即使建大表也不能实现多条数据并行查表运算操作, 打破了 SIMD 的并行性. 因此如何利用 SIMD 技术既能快速实现 ZUC-256 算法又能继承其并行性是我们整个工作的核心. 在本节中我们收集归纳了几种 ZUC-128 和 ZUC-256 的优化方法, 并将 ZUC-128 的优化方法适当推广到 ZUC-256 中, 最后分析其优缺点和可行性.

3.1.1 S_0-盒的 SIMD 实现 [26]

S_0 为 3 个 4 比特变换 P_1、P_2 和 P_3 (见表 1) 通过 3 轮 Feistel 结构迭代产生, S_0-盒的结构图具体见图 1.

表 1 $\boldsymbol{S_0}$-盒查表运算中 $\boldsymbol{P_{1/2/3}}$ 变换

Table 1 $\boldsymbol{P_{1/2/3}}$ maps used during $\boldsymbol{S_0}$ S-box algorithm

输入	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P_1	9	15	0	14	15	15	2	10	0	4	0	12	7	5	3	9
P_2	8	13	6	5	7	0	12	4	11	1	14	10	15	3	9	2
P_3	2	6	10	6	0	13	10	15	3	3	13	5	0	9	12	13

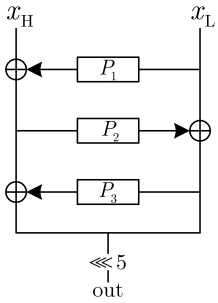


图 1 S_0-盒结构

Figure 1 Algebraic structure of S_0-box

令 x 为 8-比特字节输入, 令 x_L、x_H 分别为其低 4-比特和高 4-比特, 以下为 S_0-盒的实现详解.

(1) $t = P_1[x_L] \oplus x_H$;

(2) $\text{out}_L = P_2[\ t\ ] \oplus x_L$;

(3) $\text{out}_H = P_3[\text{out}_L] \oplus t$;

(4) $\text{out} = \text{out} \lll 5$;

(5) 返回 out.

为了利用 SIMD 技术实现 P_1、P_2 和 P_3 变换, 可以借助字节查表指令 PSHUFB, SSE 中的 _mm_shuffle_epi8 (AVX2 中的 _mm256_shuffle_epi8 以及 AVX512 中 _mm512_shuffle_epi8), 该指令接受两个 16-字节 (或 32-字节或 64-字节) 的向量: 以第一个向量 map 作为表, 第二个向量 mask 中的元素作为索引值进行查表得到一个 16-字节 (或 32-字节或 64-字节) 的向量. 例如: 令 mask =

$\{15, 14, \cdots, 0\}$, 则 _mm_shuffle_epi8(P_1, mask) = $\{9, 3, \cdots, 9\}$ 为 P_1 的反序. **算法 1** 展示了利用 AVX2 实现 S_0-盒的具体过程 (SSE 与 AVX512 类似).

算法 1 ZUC-256 S_0

输入: 32-字节向量 in

输出: 32-字节向量 out

定义: byte_low4_mask = _mm256_set_epi8(0x0F)

定义: byte_hi3_mask = _mm256_set_epi8(0xE0), byte_low5_mask = _mm256_set_epi8(0x1F)

定义: p1_mask = _mm256_set_epi32(0x09030507,0x0C000400,0x0A020F0F,0x0E000F09)

定义: p2_mask = _mm256_set_epi32(0x0209030F,0x0A0E010B,0x040C0007,0x05060D08)

定义: p3_mask = _mm256_set_epi32(0x0D0C0900,0x050D0303,0x0F0A0D00,0x060A0602)

1. 程序 S_0(in)

2. hi = _mm256_and_si256(_mm256_srli_epi32(in, 4), byte_low4_mask);

3. low = _mm256_and_si256(in, byte_low4_mask);

4. r = _mm256_xor_si256(hi, _mm256_shuffle_epi8(p1_mask, low));

5. s = _mm256_xor_si256(low, _mm256_shuffle_epi8(p2_mask, r));

6. t = _mm256_xor_si256(r, _mm256_shuffle_epi8(p3_mask, s));

7. tmp = _mm256_or_si256(s, _mm256_slli_epi32(t, 4));

8. low = _mm256_and_si256(_mm256_srli_epi32(tmp, 3), byte_low5_mask);

9. hi = _mm256_and_si256(_mm256_slli_epi32(tmp, 5), byte_hi3_mask);

10. out = _mm256_or_si256(low, hi);

11. 返回 out.

需要说明的是 _mm256_set_epi32 指令只允许 8 个 32-比特的字作为输入, 按先高位后低位的顺序排列, **算法 1** 中的 p1_mask、p2_mask 和 p3_mask 应当分别输入两次 P_1、P_2 和 P_3 的值 (AVX512 需要输入四次), 即 p1_mask = _mm256_set_epi32(0x09030507, 0x0C000400, 0x0A020F0F, 0x0E000F09, 0x09030507, 0x0C000400, 0x0A020F0F, 0x0E000F09), 因此在实际的 C 代码中需要输入两次. 在这里由于篇幅原因, **算法 1** 中只输入了一次, 下文中情形类似.

3.1.2 S_1-盒的 SIMD 实现 [26]

从 2.2 节可知, S_1 是基于有限域 $\mathbb{GF}(256)$ 上的逆函数构造的, 与分组密码 AES 的 S-盒类似, 它们之间仿射等价. 记 $\mathbb{F}_{\mathsf{AES}}$ 表示模本原多项式 $f_{\mathsf{AES}}(x) = x^8 + x^4 + x^3 + x + 1$ 的有限域 $\mathbb{GF}(256)$, AES 算法的 S-盒 (记为 S_{AES}) 的构造为: $S_{\mathsf{AES}}(x) = A \cdot x^{-1} + D$, 其中 A 为二元矩阵, D = 0x63; 同时记 $\mathbb{F}_{\mathsf{ZUC}}$ 表示模本原多项式 $f_{\mathsf{ZUC}}(x) = x^8 + x^7 + x^3 + x + 1$ 的有限域 $\mathbb{GF}(256)$, S_1-盒的构造为: $S_1(x) = M \cdot x^{-1} + B$, 其中 B = 0x55, M 为二元矩阵. 二元矩阵 A 和 M 具体如下:

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}, \quad M = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{pmatrix}.$$

由于 S_1-盒的构造和 S_{AES}-盒的构造是仿射等价的, 因此, 存在一个域 $\mathbb{F}_{\mathsf{ZUC}}$ 到域 $\mathbb{F}_{\mathsf{AES}}$ 的同构映射 ϕ,

$$\begin{aligned}\phi : \mathbb{F}_{\mathsf{ZUC}} &\longrightarrow \mathbb{F}_{\mathsf{AES}}\\ x &\longmapsto Kx,\end{aligned}$$

其中, K 为一个二元可逆矩阵, 文献 [26] 中给出了一种计算矩阵 K 的方法. 因此在同构映射 ϕ 下对任意的 $x \in \mathbb{F}_{\mathsf{ZUC}}$ 都存在唯一的 $y = Kx \in \mathbb{F}_{\mathsf{AES}}$, 通过利用 AESENCLAST 指令来计算 $y^{-1} \in \mathbb{F}_{\mathsf{AES}}$, 最后乘以 K^{-1} 便可得到 $x^{-1} \in \mathbb{F}_{\mathsf{ZUC}}$, 这样便可以通过 AES-NI 指令集来达到实现并行查 S_1 的目的. 首先将 S_{AES}-盒转化为:

$$S_{\mathsf{AES}}[x] = A \cdot (x^{-1} \bmod f_{\mathsf{AES}}(x)) \oplus D.$$

同时, S_1-盒:

$$S_1[x] = M \cdot (x^{-1} \bmod f_{\mathsf{ZUC}}(x)) \oplus B.$$

在同构映射 ϕ 下, ZUC-256 算法的加密 S_1-盒可以等价的构造为:

$$S_1[x] = T \cdot (S_{\mathsf{AES}}[K \cdot x] \oplus D) \oplus B, \tag{2}$$

其中 $T = M \cdot K^{-1} \cdot A^{-1}$, 根据文献 [26] 给出的方法计算矩阵 K, K 共有 8 种选择, 在这里选择以下的 K 值, 同时矩阵 T 也可以被同步计算出来.

$$K = \begin{pmatrix} 1&1&1&0&0&0&1&0\\ 0&0&1&0&1&1&1&1\\ 0&0&0&0&0&0&1&1\\ 0&1&1&0&0&1&0&0\\ 0&1&0&0&0&0&0&0\\ 0&0&1&1&0&0&0&1\\ 0&1&1&1&0&1&1&0\\ 0&1&0&0&1&0&1&0 \end{pmatrix}, \quad T = \begin{pmatrix} 1&0&1&0&1&0&0&0\\ 0&0&1&0&0&0&0&1\\ 1&1&1&0&0&1&1&1\\ 1&0&1&1&1&0&0&1\\ 1&0&1&0&1&1&1&1\\ 0&1&1&0&0&1&1&0\\ 1&0&1&0&0&0&1&0\\ 1&0&1&0&1&0&0&1 \end{pmatrix}.$$

公式(2)中的 S_1-盒的计算可以分解为矩阵乘法、S_{AES}-盒查表以及异或运算, 而矩阵乘法可以转换为 2 次查表 (4 进 8 出) 和 1 次异或运算, 以下为 S_1-盒的实现详解.

(1) 计算 y, K_1, K_2.

$$\begin{aligned} y &= K \cdot x = K(x)\\ &= K(x \ \&\ \text{0x0F}) \oplus K(x \ \&\ \text{0xF0})\\ &= K_1(x \ \&\ \text{0x0F}) \oplus K_2((x \gg 4) \ \&\ \text{0x0F});\\ K_1 &= \{\text{0x00,0x01,0xD9,0xD8,0x6B,0x6A,0xB2,0xB3,0x60,0x61,0xB9,0xB8,0x0B,0x0A,0xD2,0xD3}\};\\ K_2 &= \{\text{0x00,0x82,0x4A,0xC8,0xC7,0x45,0x8D,0x0F,0x26,0xA4,0x6C,0xEE,0xE1,0x63,0xAB,0x29}\}; \end{aligned}$$

(2) 计算 $z = S_{\mathsf{AES}}[y] \oplus D$, 使用 AESENCLAST 指令时需要注意行移位变换;

(3) 计算 w, T_1, T_2.

$$\begin{aligned} w &= T \cdot z \oplus B = T(z) \oplus B;\\ &= (T(z \ \&\ \text{0x0F}) \oplus B) \oplus T(z \ \&\ \text{0xF0})\\ &= T_1(z \ \&\ \text{0x0F}) \oplus T_2((z \gg 4) \ \&\ \text{0x0F});\\ T_1 &= \{\text{0x55,0x88,0x71,0xAC,0xAA,0x77,0x8E,0x53,0x5D,0x80,0x79,0xA4,0xA2,0x7F,0x86,0x5B}\};\\ T_2 &= \{\text{0x00,0x99,0x34,0xAD,0x74,0xED,0x40,0xD9,0x9E,0x07,0xAA,0x33,0xEA,0x73,0xDE,0x47}\}. \end{aligned}$$

AESENCLAST 指令用于实现分组算法 AES 的最后一轮函数, 包括: 行移位变换、字节替代即 S_{AES}、轮密钥加. 对上述步骤 (2) 中的 y 进行逆行移位变换以及将轮密钥置为 D, 利用 AESENCLAST 指令便可得到 z 进而计算出 S_1. **算法 2** 为 S_1-盒用 AVX2 实现的具体过程 (SSE 与 AVX512 类似).

算法 2 ZUC-256 S_1

输入: 32-字节向量 input

输出: 32-字节向量 output

定义: k_mask1 = _mm256_set_epi32(0xD3D20A0B,0xB8B96160,0xB3B26A6B,0xD8D90100)

定义: k_mask2 = _mm256_set_epi32(0x29AB63E1,0xEE6CA426,0x0F8D45C7,0xC84A8200)

定义: t_mask1 = _mm256_set_epi32(0x5B867FA2,0xA479805D,0x538E77AA,0xAC718855)

定义: t_mask2 = _mm256_set_epi32(0x47DE73EA,0x33AA079E,0xD940ED74,0xAD349900)

定义: aes_row_invshift = _mm256_set_epi32(0x0306090C,0x0F020508,0x0B0E0104,0x070A0D00)

定义: aes_cancelkey = _mm_set_epi8(0x63)

1. **程序** S_1(input)

2. low = _mm256_shuffle_epi8(k_mask1, _mm256_and_si256(input, byte_low4_mask));

3. hi = _mm256_shuffle_epi8(k_mask2, _mm256_and_si256(_mm256_srli_epi32(input, 4), byte_low4_mask));

4. tmp = _mm256_shuffle_epi8(aes_row_invshift, _mm256_xor_si256(hi, low));

5. tmp = _mm_aesenclast_si128(tmp, aes_cancelkey);

6. low = _mm256_shuffle_epi8(t_mask1, _mm256_and_si256(tmp, byte_low4_mask));

7. hi = _mm256_shuffle_epi8(t_mask2, _mm256_and_si256(_mm256_srli_epi32(tmp, 4), byte_low4_mask));

8. output = _mm256_xor_si256(low, hi);

9. **返回** output.

注意在 **算法 2** 中步骤 5 和 6 只允许接收 16-字节的输入, 这是因为 Intel 目前只给出 16-字节的 AES-NI 指令集, 因此当使用 AVX2 和 AVX512 指令集时分别需要调用 2 次和 4 次 AESENCLAST 指令, 单次查 S_1-盒的效率会比 SSE 低, 但是对比经典查表方法这依然是非常高效的.

3.1.3 延迟模约 [12]

在计算 LFSR 时, 为了保证公式(1)的正确性, 需要对(1)式中的每次乘法和加法结果进行模约运算. 文献 [12] 中提出延迟模约方法对单密钥模式下的 ZUC-256 密钥流生成算法进行优化, 其主要思想是: 模数 $p = 2^{31} - 1$ 对于加法模约运算是非常快速的, 只需要用到与运算和移位运算 (对任意 $x, y, z \in \mathbb{H}_p, k \in \mathbb{Z}$, $k = x + y$, 则 $z = x + y \bmod p = (k \ \&\ \text{0x7FFFFFFF}) + (k \gg 31)$), 但乘以一个 2 的幂的乘法模约运算需要用到循环运算, 这对软件优化来说需要更多开销, 如何减少乘法模约运算?

令 $k \in \mathbb{Z}$, 则 $k = (1 + 2^8)s_0 + 2^{20}s_4 + 2^{21}s_{10} + 2^{17}s_{13} + 2^{15}s_{15}$ 的计算是在整环 $\mathbb{Z}$ 上的乘法和加法, 没有模约处理, 很明显 $k < 2^{22} \cdot 2^{31} = 2^{53}$; 然后计算 $k' = (k \ \&\ \text{0x7FFFFFFF}) + (k \gg 31) < 2^{32} - 2$, 只需要对 k' 再进行一次加法模约处理可以得到 $k \bmod p$, 因此计算公式(1)整个过程只需要两次加法模约, 在单密钥模式下, 如果需要, 还可以使用汇编语言提升速度.

然而, 若在公式(1)中应用延迟模约方法, 需要将 32 位上的加法和乘法操作对应转换到 64 位上, 这对于多密钥模式来说需要较大开销 (例如: 对于一个 8×32-比特的向量, 将其转换为 8×64-比特的向量, 则需要两个 256-比特的向量进行存储以及其它操作, 最后再将其转为一个 8×32-比特的向量), 经过测试, 在多密钥模式应用延迟模约比乘法模约运算开销更大. 因此, 延迟模约技术适合应用到单密钥模式 (如果不使用汇编, 对比乘法模约, 提升幅度也并不高), 在多密钥模式下, 使用乘法模约开销会更少.

3.2 MAC 生成算法的 SIMD 实现

利用 SIMD 技术并行实现 MAC 生成算法首先需要解决两个问题, 一是明文消息的比特判断, 二是密钥流每次只能进行 "1-比特" 运算. 在本节中, 我们给出了 MAC 生成算法的两种 SIMD 实现方法.

3.2.1 利用比较指令生成 MAC 的 SIMD 实现

2.3 节描述了 MAC 的生成, 从其描述步骤 (3.2) 可知, 每次迭代都需要对明文消息的每 1-比特进行判断. 然而, ZUC-256 算法的基本单元都是 32-比特的字, 按比特判断不仅会降低软件整体性能, 而且不利于使用 SIMD 技术并行处理多条数据, 因此我们的目标是尽量避免明文消息的比特判断以及每次迭代都是按字计算而不是按比特计算. 为了方便描述, 在本文中我们主要针对认证标签长度为 32-比特的 MAC 生成 (64-比特 $= 2 \times 32$ 比特, 128-比特 $= 4 \times 32$ 比特).

假设生成的密钥比特流为 $z_0, z_1, \cdots$, 明文比特序列为 $m_0, m_1, \cdots$, 则 2.3 节中 MAC 生成描述步骤 (3) 中的 W_i 可以等价转化为 $W_i = m_i z_i \| m_i z_{i+1} \| \cdots \| m_i z_{i+31}$. 以下为 MAC 生成算法的前 32 次迭代的一个等价转换.

(1) 令 $\overline{Z_i} = z_i \| z_{i+1} \| \cdots \| z_{i+31}$, 其中 $0 \le i < 32$;

(2) 记 $T_i = m_i \cdot \overline{Z_i} = m_i z_i \| m_i z_{i+1} \| \cdots \| m_i z_{i+31}$;

(3) 则 $\text{Tag} = \text{Tag} \oplus T_0 \oplus T_1 \oplus \cdots \oplus T_{31}$.

在单密钥模式下, 上述的 T_i 在本质上依旧避免不了对明文序列的判断, 但是在多密钥模式下, 我们可以利用比较指令 PCMPGTD 来很好的规避这一问题, **算法 3** 展示了利用 AVX2 中的 PCMPGTD 指令生成 32-比特 Tag 的部分过程, 对应 2.3 节中的描述步骤 (3).

算法 3 ZUC-256 MAC 生成算法-1

输入: 32-字节向量密钥流组 *vecz, $8 \times L$ 个 32-比特明文数组 *msg

输出: 32-字节向量 vtag

定义: V2ZERO = _mm256_setzero_si256()

1. 程序 MAC_Generate(vecz, msg)

2. i 从 0 到 $L-1$ 依次执行:

3. r = _mm256_loadu_si256(msg + $8 \times i$);

4. j 从 0 到 31 依次执行:

5. s = _mm256_slli_epi32(r, j); t = _mm256_cmpgt_epi32(V2ZERO, s);

6. u = _mm256_srli_epi32(vecz[$i+2$], $32-j$);

7. v = _mm256_slli_epi32(vecz[$i+1$], j); w = _mm256_or_si256(u, v);

8. vtag = _mm256_xor_si256(vtag, _mm256_and_si256(w, t));

9. vtag = _mm256_xor_si256(vtag, tmp); tmp = vtag;

10. 返回 vtag.

算法 3 的主要思想是利用 _mm256_cmpgt_epi32(a, b) 的输出结果, 该指令接受两个 256-比特的向量, 按有符号的 32-比特进行比较, 大于输出 0xFFFFFFFF, 否则输出 0; 因此将明文消息字进行左移, 然后与 0 进行比较, 如果移位后的明文消息字最高位为 1, 则为负数, 经过比较指令后输出 0xFFFFFFFF, 再经过与运算就可以取出 W_i.

3.2.2 利用无进位乘法指令生成 MAC 的 SIMD 实现

算法 3 中虽然利用 PCMPGTD 比较指令规避了明文序列的判断, 但是在步骤 7 中的或运算等价于在密钥比特流上按比特滑动, 这类似于字循环操作, 并不利于软件优化; 除此之外, 在步骤 4 中仍然需要执行 32 次才能处理一个明文消息字, 换句话说, 这在本质上是按比特计算, 依然没有达到真正意义上的按字计算. 文献 [12] 给出一种方法, 在单密钥模式下利用无进位乘法指令 PCLMULQDQ 来快速生成 ZUC-128 的 MAC, 可以每次迭代处理一个明文消息字. 与 ZUC-128 流密码中的 MAC 生成算法相比, ZUC-256 在算法结构上并没有太大的改变, 因此我们可以将文献 [12] 中的方法推广到 ZUC-256 的 MAC 生成算法的多密钥模式中, 并优化了其方法中的某些步骤, 以此加速其实现进程.

假设生成的密钥比特流为 $z_0, z_1, \cdots$, 明文比特序列为 $m_0, m_1, \cdots$, 则密钥比特流和明文序列可以看做是一个 $\mathbb{F}_2[x]$ 上的线性泛哈希函数的输入. 2010 年, Intel 在他们的 32 纳米处理器家族上引进了一

种新的指令 PCLMULQDQ; 令人感到惊喜的是, 该指令可以计算两个 64-比特数的无进位乘法乘积, 即计算两个在 $\mathbb{F}_2[x]$ 上次数最多为 63-次的二元多项式的乘积 (最多为 126-次的多项式). 在实现性能方面, PCLMULQDQ 能够在 14 个时钟圈内执行, 这远快于经典的 $\mathbb{F}_2[x]$ 上的多项式乘法运算. 因此我们可以利用无进位乘法指令 PCLMULQDQ 来计算 MAC 生成. 下面我们展示如何利用 PCLMULQDQ 指令快速生成 MAC, 首先我们将 2.3 节 MAC 生成描述步骤 (3) 中前 32 次计算等价转换为另一种描述.

(1) 令 $r = (m_{31}, m_{30}, \cdots, m_0)^{\mathrm{T}}$;

(2) 令 矩阵

$$K = \begin{pmatrix} z_{31} & z_{30} & \cdots & z_0 \\ z_{32} & z_{31} & \cdots & z_1 \\ \vdots & \vdots & \ddots & \vdots \\ z_{62} & z_{61} & \cdots & z_{31} \end{pmatrix};$$

(3) 则 $\mathrm{Tag} = \mathrm{Tag} \oplus (K \cdot r)$.

注意 $K \cdot r$ 是在域 $\mathbb{F}_2$ 上进行的, 即操作运算中的加法为异或运算. 令 m, p, p_1, p_2, $q \in \mathbb{F}_2[\mathrm{x}]$, $m = m_0 + m_1 x + \cdots + m_{31} x^{31}$ 为 $\mathbb{F}_2[x]$ 上的一个 31-次多项式, 其序和第一个明文消息字的序相反; $p_1 = z_{31} + z_{30} x + \cdots + z_0 x^{31}$ 为第二个密钥流字作为 $\mathbb{F}_2[x]$ 上的一个 31-次多项式; $p_2 = z_{63} + z_{62} x + \cdots + z_{32} x^{31}$ 为下一个密钥流字; $p = p_1 x^{32} + p_2$ 为两个密钥流字构成的 63-次多项式; $q = mp = q_2 x^{64} + q_1 x^{32} + q_0$ 为多项式 m 和 p 的乘积, 我们将证明 q_1 (一个 32-比特的字, 作为多项式的系数) 和 $K \cdot r$ 是等价的.

$$\begin{aligned} q = mp &= \left(\sum_{i=0}^{31} m_i x^i\right) \times \left(\sum_{i=0}^{63} z_i x^{63-i}\right) \\ &= \begin{pmatrix} 0 & 0 & \cdots & 0 \\ z_0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ z_{30} & \cdots & z_0 & 0 \end{pmatrix} \begin{pmatrix} m_{31} \\ m_{30} \\ \vdots \\ m_0 \end{pmatrix} \otimes \begin{pmatrix} x^{95} \\ x^{94} \\ \vdots \\ x^{64} \end{pmatrix} + \begin{pmatrix} z_{31} & z_{30} & \cdots & z_0 \\ z_{32} & z_{31} & \cdots & z_1 \\ \vdots & \vdots & \ddots & \vdots \\ z_{62} & z_{61} & \cdots & z_{31} \end{pmatrix} \begin{pmatrix} m_{31} \\ m_{30} \\ \vdots \\ m_0 \end{pmatrix} \otimes \begin{pmatrix} x^{63} \\ x^{62} \\ \vdots \\ x^{32} \end{pmatrix} + \begin{pmatrix} z_{63} & z_{62} & \cdots & z_{32} \\ 0 & z_{63} & \cdots & z_{33} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & z_{63} \end{pmatrix} \begin{pmatrix} m_{31} \\ m_{30} \\ \vdots \\ m_0 \end{pmatrix} \otimes \begin{pmatrix} x^{31} \\ x^{30} \\ \vdots \\ 1 \end{pmatrix}. \end{aligned}$$

因此 MAC 的生成运算可以等价转化为 $\mathbb{F}_2[x]$ 上的多项式乘法. 在实际的软件实现中, 首先将消息字的反序 (将反序结果转换为 64-比特, 高 32 位为 0) 和两个 32-比特密钥流字作为 PCLMULQDQ 指令的输入, 取出其运算结果中的 q_1 便可得到 $K \cdot r$. 对后续的所有明文消息字重复同样的步骤直到我们获得最终的 MAC 码, 达到了每次按字计算而非比特计算的快速软件实现目标. **算法 4** 为 AVX2 中的 PCLMULQDQ 指令生成 32-比特 Tag 的实现过程.

在 **算法 4** 中, PCLMULQDQ 指令需要输入两个 128-比特向量以及一个控制选择位, 该指令根据控制选择位每次选取两个 128-比特向量中某两个 64-比特数计算得到一个 128-比特向量, 这在 AVX2 和 AVX512 中需要多次调用才能得到结果, 其效率会比 SSE 低. 除此之外, 在步骤 3 中还需要对明文序列字进行反序处理, 目前 SIMD 技术在 x86 平台上还没有提供比特反序的指令, 如果未来 Intel 提供比特反序的指令, 利用 PCLMULQDQ 指令会有更进一步的优势.

4 方案优化与试验结果

在本节中我们将第 3 节中的技术应用到 ZUC-256 算法中, 并对其中一些技术方法做进一步的优化; 在 4.4 节中我们对比几种不同方法实现 ZUC-256 流密码算法的软件性能, 并分析其结果. 我们已将完整的 C 实现代码 (AVX512 版本) 公开发布在 GitHub 网站上: https://github.com/Nonights/ZUC256.

4.1 S-盒优化

在第 3 节中, **算法 1** 和**算法 2** 的输入输出均为 32-字节的向量 (针对 AVX2), 然而回顾 ZUC-256 算法描述中对于查表操作 $S(Y_3||Y_2||Y_1||Y_0) = S_0(Y_3)||S_1(Y_2)||S_0(Y_1)||S_1(Y_0)$, 即在单密钥模式下, S_0-盒只

算法 4 ZUC-256 MAC 生成算法-2

输入: 32-字节向量密钥流组 vecz, $8 \times L$ 个 32-比特明文数组 msg

输出: 32-字节向量 vtag (8 个并行的 32-比特 Tag)

1. 程序 MAC_Generate(vecz, msg)

2. i 从 0 到 $L-1$ 依次执行:

3. r = Word_reverse(_mm256_loadu_si256(msg + $8 \times i$)); /* 对每一路消息字进行反序处理 */

4. $s[4]$ = Expandtovec128(r); /* 将 r 的每路数据扩展为 64 比特并转换成 4 个 SSE 中的向量 */

5. $w[4]$ = Storetovec128(vecz[$i+1$], vecz[$i+2$]); /* 将两个密钥流向量转换成 4 个 SSE 中的向量 */

6. j 从 0 到 3 依次执行:

7. $u[j]$ = _mm_clmulepi64_si128(s[j], w[j], 0x00);

8. $v[j]$ = _mm_clmulepi64_si128(s[j], w[j], 0x11);

9. tmp = Loadtovec256($u[4]$, $v[4]$); /* 取出每路中的 q_1 并生成一个 32-字节向量 */

10. tmp = _mm256_xor_si256(tmp, temp); temp = tmp; /* 更新 tmp, temp 的初始值置为 0 */

11. vtag = _mm256_xor_si256(_mm256_xor_si256(vecz[0], tmp), vecz[$L+1$]);

12. 返回 vtag.

对字 Y 的奇字节位进行查表, 而 S_1-盒只对字 Y 的偶字节位进行查表; 换句话说, 在多密钥 8×32 模式下, 对于一个 32-字节的向量 vec_s, S_0 和 S_1 只需要分别对其奇字节位和偶字节位进行查表. 第一种方法是利用 **算法 1** 和 **算法 2** 先分别计算出 $S_0(\mathrm{vec}_s)$ 与 $S_1(\mathrm{vec}_s)$, 然后再分别取出奇字节和偶字节, 最后再移位异或得到 $S(\mathrm{vec}_s)$; 然而以这种方法查表计算, 会浪费很多开销, 因为 S_0 不需要对 vec_s 中的偶字节位进行查表, 同理 S_1 也不需要对 vec_s 中的奇字节位进行查表. 第二种方法如下: 在单密钥模式下, 假设 $R_1 = S(X_3||X_2||X_1||X_0) = S_0(X_3)||S_1(X_2)||S_0(X_1)||S_1(X_0)$, $R_2 = S(Y_3||Y_2||Y_1||Y_0) = S_0(Y_3)||S_1(Y_2)||S_0(Y_1)||S_1(Y_0)$, 则令 $R'_1 = S_0(X_3||Y_3||X_1||Y_1) = S_0(X_3)||S_0(Y_3)||S_0(X_1)||S_0(Y_1)$, $R'_2 = S_1(X_2||Y_2||X_0||Y_0) = S_1(X_2)||S_1(Y_2)||S_1(X_0)||S_1(Y_0)$, 最后再通过移位、与和异或便可得到 R_1 和 R_2, 避免不必要的开销. 除此之外, 在使用第二种方法实现 S 时, 还可以利用混合指令 VPBLENDVB 和移位指令 VPSLLD (或 VPSRLD) 进一步优化实现, **算法 5** 为 2.2 节里描述步骤 (3) 中 R 的 AVX2 实现 (SSE 和 AVX512 类似).

算法 5 ZUC-256 R

输入: 32-字节向量 l_1, l_2

输出: 32-字节向量 r_1, r_2

定义: even_byte_mask = _mm256_set1_epi32(0x00FF00FF)

1. 程序 $R(l_1, l_2)$

2. $a = S_0$(_mm256_blendv_epi8(l_1, _mm256_srli_epi32(l_2, 8), even_byte_mask));

3. $b = S_1$(_mm256_blendv_epi8(_mm256_slli_epi32(l_1, 8), l_2, even_byte_mask));

4. r_1 = _mm256_blendv_epi8(a, _mm256_srli_epi32(b, 8), even_byte_mask);

5. r_2 = _mm256_blendv_epi8(_mm256_slli_epi32(a, 8), b, even_byte_mask);

6. 返回 r_1, r_2.

4.2 向量转换优化

目前, Intel 只提供了 AESENCLAST 和 PCLMULQDQ 的 SSE 指令, 即只允许 128-比特的运算操作, 因此当使用 AVX2 和 AVX512 时分别需要调用 2 次和 4 次相应的 SSE 指令. 例如在 AVX2 指令集中, 对于一个 256-比特的输入向量 vec, 当调用 AESENCLAST 指令时, 需要将 vec 转换为两个 128-比特的向量; 这里有两种转换方法, 一种是利用 memcpy 命令或者利用指针进行强制转换, 然而这种方法不是并行处理的. 另一种方法是利用转换、提取和插入指令, 这种方法可以保证算法过程的并行性; 其次对比第一种方法, 后者在多次调用中性能会有明显提升. **算法 6** 为 **算法 1** 中步骤 5 的具体 AVX2 实现.

算法 6 ZUC-256 USE_AES-NI

输入: 32-字节向量 tmp

输出: 32-字节向量 tmp

1. **程序** USE_AES-NI(tmp)

2. _mm256_storeu2_m128i(temp, temp + 1, tmp); /* temp 为两个 128-比特的临时向量组 */

3. temp[0] = _mm_aesenclast_si128(temp[0], aes_concelkey);

4. temp[1] = _mm_aesenclast_si128(temp[1], aes_concelkey);

5. tmp = _mm256_loadu2_m128i(temp, temp + 1);

6. **返回** tmp.

算法 6 中的方法依然可以推广到 AVX512, 在 AVX512 中需要用到四次 512-比特到 128-比特向量转换、四次 AESENCLAST 指令调用和四次 128-比特到 512-比特向量转换, 性能相对会进一步下降. 利用 SIMD 技术实现 MAC 生成时, 调用 PCLMULQDQ 指令和 AESENCLAST 情形类似, 但是需要注意明文消息的注入 (每一路 32-比特消息需要进行反序处理并扩展为 64-比特, 高 32-比特为 0) 和密钥流字的顺序.

4.3 LFSR 层的移位操作优化

除了算法本身的优化以外, 还可以通过增加代码的复杂度减少 LFSR 层的移位操作来提升性能. 在密钥流生成阶段, 每一轮迭代对 BR 层、FSM 层和 LFSR 层的状态进行更新, 当明文消息流足够长的时候, 代码中 for 循环也在增加. 注意到 LFSR 层, 每次只对最后一个寄存器的值进行更新, 而其他值都是由后一个寄存器的值覆盖更新, 在 16 个迭代中所有值全部被更新, 因此可以通过多轮连续迭代减少移位操作. 例如在密钥流生成阶段, 在一个循环内做 16 次迭代可以最大程度的减少移位操作.

4.4 试验结果

在本小节中, 我们将测试不同方法的软件实现性能, 分为密钥流生成和 MAC 生成两个大类 (见表 2); 注意我们以文献 [4] 中的参考代码为基准, 其余实现方法都是在此基础上进行优化. 我们对比文献 [12] 中的延迟模约方法、文献 [26] 中的 SIMD 技术和本文的优化 SIMD 技术分别实现 ZUC-256 密钥流生成算法的软件性能, 以及对比文献 [26] 中的无进位乘法方法、本文中的比较指令 (方法 1) 和无进位乘法指令 (方法 2) 的 SIMD 技术分别生成 MAC 的软件性能.

表 **2** 对比测试版本

Table 2 Eight versions of test

分类	来源	技术方法
密钥流生成优化方法	文献 [4]	–
	文献 [12]	延迟模约
	文献 [26]	SIMD 技术
	本文	优化 SIMD 技术
MAC 生成优化方法	文献 [4]	–
	文献 [12]	无进位乘法方法
	本文方法 1	比较指令的 SIMD 技术
	本文方法 2	无进位乘法的 SIMD 技术

我们分别在 2 个不同处理器平台上对 ZUC-256 的软件实现性能进行评估, 一个是 Intel(R) Core(TM) i3-6100 CPU @3.70GHz, 该平台有 4GB 内存, 系统类型为 32 位 Win7 操作系统, 记为 Platform1; 另一个平台是 Intel(R) Xeon(R) Gold 6128 CPU @3.40 GHz, 128 GB 内存, 64 位 Win10 操作系统, 记为 Platform2. Platform1 和 Platform2 分别使用 Visual Studio 2015 软件和 Visual Studio 2017 软件进行

性能测试, 同时两个平台除了算法本身的优化外, 没有对软件进行额外的优化设置, 均使用的是软件默认设置. Platform1 和 Platform2 中运行软件的解决方案配置均采用 Release 模式, 前者解决方案平台使用 x86, 后者使用 x64.

表 2 中的测试方法均使用 C 代码实现, 没有使用汇编语言; 其中文献 [4] 和文献 [12] 中的四种方法均为在单密钥模式下的软件实现, 而剩余四种方法利用 SIMD 技术并行实现. 除此之外, Platform1 中使用的 SIMD 技术均为 AVX2, Platform2 中的 SIMD 技术均为 AVX512. 图 2、图 3、图 4 和图 5 分别展示了两个不同平台上 ZUC-256 密钥流生成和 MAC 生成的各个实现方法的软件性能. 评估 ZUC-256 密钥流生成算法和 MAC 生成算法的软件实现性能指标是算法每秒钟处理数据的比特数, 即 bps; 在本文中, 1 Gbps = 10^9 bps. 在 Platform1 中, 我们将 LFSR 层的移位操作优化技巧应用到本文的 3 个实现方法中, 和没有增加代码复杂度的实现方法相比, 速度提升了大约 15%. 从图 2–图 5 可以发现, 密钥流生成算法和 MAC 生成算法的最优软件实现性能结果均为 AVX512 版本, 密钥流生成算法的 AVX512 版本实现性能可以达到 21 Gbps, 相较于目前已知的最优实现方法 [26] 性能提升了 30%, 而 AVX2 版本的实现性能则提升了 56%, 主要原因在于编译软件的不断智能化, Visual Studio 2017 对于代码中的类似 LFSR 层的移位操作会自行进行优化 (注: 由于文献 [26] 的参考文献中给出的完整实现代码所在网址无法打开, 我们对比的结果是通过利用文献 [26] 中给出的方法然后自行编写的代码运行得出的, 但文献 [26] 中结论表示 AVX512 版本比 AVX2 的实现性能稍差, 这与理论逻辑不符, 并且我们的运行结果也验证了这一点). MAC 生成算法同文献 [4] 中的无优化实现方法相比性能提高了 20 倍. 在表 2 中我们只列举了一部分优化方法的组合对比, 然而某些不在表 2 中的其他的优化方法组合可能会有更进一步的速度提升, 在这里我们就不对所有可能的组合进行一一测试.

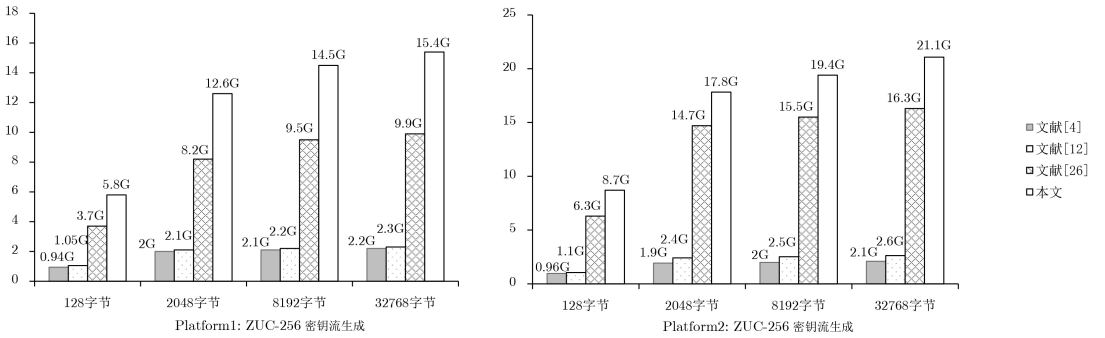


图 2 四种 ZUC-256 密钥流生成实现方法在两个平台上的性能表现

Figure 2 Performance of four different methods of ZUC-256 crypt on two platforms

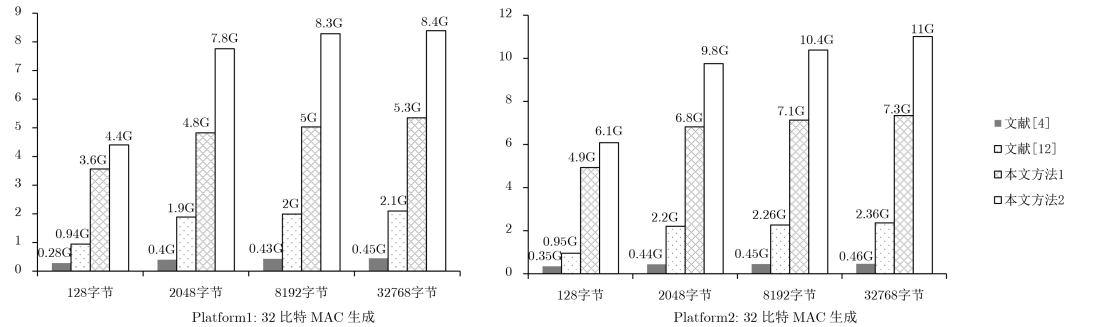


图 3 四种 32 比特 MAC 生成实现方法在两个平台上的性能表现

Figure 3 Performance of four different methods of 32-bits MAC generation on two platforms

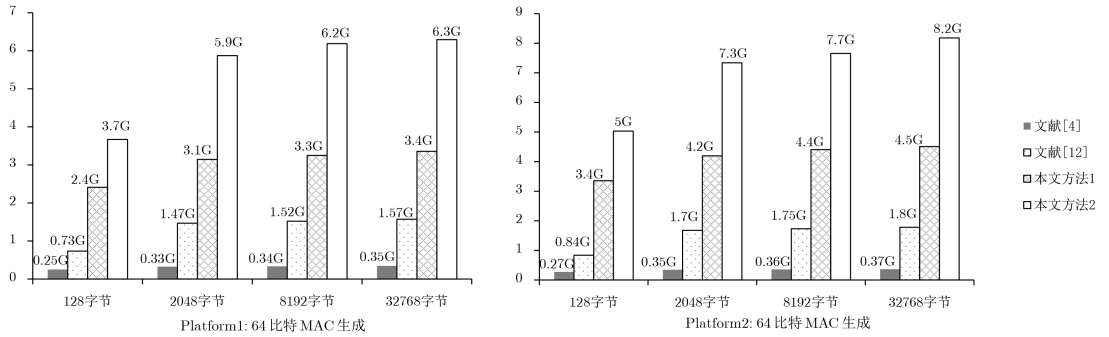


图 4 四种 64 比特 MAC 生成实现方法在两个平台上的性能表现

Figure 4 Performance of four different methods of 64-bits MAC generation on two platforms

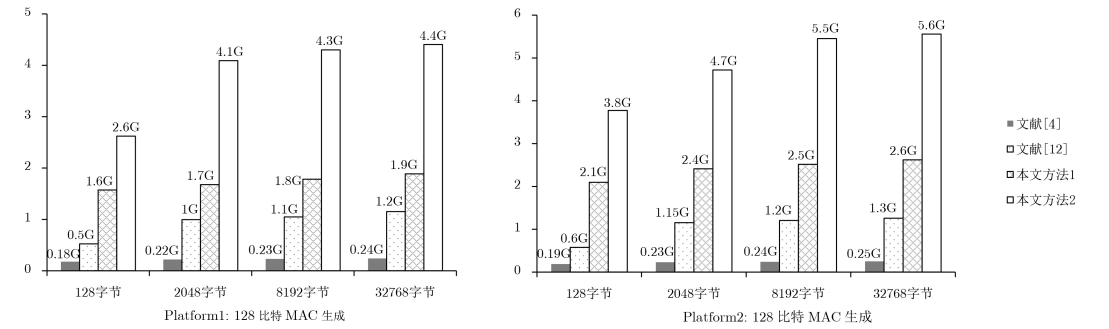


图 5 四种 128 比特 MAC 生成实现方法在两个平台上的性能表现

Figure 5 Performance of four different methods of 128-bits MAC generation on two platforms

5 结论

在本文中, 我们给出了 ZUC-256 序列密码算法的整体快速软件实现方法, 包括密钥流生成算法和 MAC 生成算法. 密钥流生成算法是基于文献 [26] 中提出的方法基础上快速实现的, 我们一方面优化了 S-盒的查表操作, 避免了不必要的查表开销; 另一方面, 在调用 AES-NI 指令集时, 我们还对向量转换方式进行优化, 保证了 SIMD 实现过程的并行性. 实验结果表明, 我们的优化方案和文献 [26] 相比软件实现性能具有较大优势, 在 Intel(R) Core(TM) i3-610 处理器和 Intel(R) Xeon(R) Gold 6128 处理器上的软件实现性能分别提升了 56% 和 30%, 并且在后者处理器上的软件实现性能可以达到 21 Gbps, 超过了 5G 系统中的下行速度要求. 由于 ZUC-128 算法和 ZUC-256 算法中的 MAC 生成算法大体结构是一样的, 文献 [12] 中构造的无进位乘法方法依然可以应用到 ZUC-256 的 MAC 生成算法中; 同无优化的 MAC 生成算法的软件实现性能相比, 我们利用无进位乘法指令并行实现 MAC 的方法具有非常明显的优势. 但是, 在密钥流生成算法和 MAC 生成算法中用到的 AESENCLAST 指令和 PCLMULQDQ 指令目前只允许输入 128-比特的向量 (SSE), 因此使用 AVX2 和 AVX512 指令集需要多次调用, 性能会有所降低. Intel 在 2019 年 5 月发布的白皮书中提出未来要扩展现有的指令集, 尤其是都规划了 AESENCLAST 指令和 PCLMULQDQ 指令的 AVX2 和 AVX512 版本 [32]. 等到这些指令集正式发布以后, 相信 ZUC-256 算法的软件实现性能会有更进一步的提升.

参考文献

[1] 3GPP. 3rd Generation Partnership Project TS 26.233: Technical specification group radio access network[R/OL]. Evolved Universal Terrestrial Radio Access (E-UTRA). http://www.3gpp.org

[2] ETSI/SAGE. Specification of the 3GPP confidentiality and integrity algorithms 128-EEA3 & 128-EIA3[R/OL].

Document 4: Design and Evaluation Report. 2011.
https://www:gsma:com/aboutus/wp-content/uploads/2014/12/EEA3 EIA3 Design Evaluation v20:pdf
[3] 3GPP SA3. TR 33.841 Study on supporting 256-bit algorithms for 5G[R/OL]. 2018.
https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3422
[4] 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]
[5] PATRIK E, THOMAS J, ALEXANDER M, et al. A new SNOW stream cipher called SNOW-V[J/OL]. IACR Cryptology ePrint Archive, 2018: 2018/1143. http://eprint.iacr.org//2018/1143.pdf
[6] ITU. Minimum requirements related to technical performance for IMT-2020 radio interface(s)[EB/OL]. Version 1.0, 2017. https://www.itu.int/pub/R-REP-M. 2410-2017
[7] KITSOS P, SKLAVOS N, SKODRAS A N. An FPGA implementation of the ZUC stream cipher[C]. In: Proceedings of 2011 14th Euromicro Conference on Digital System Design. IEEE, 2011: 814–817. [DOI: 10.1109/DSD.2011.109]
[8] KITSOS P, SKLAVOS N, PROVELENGIOS G, et al. FPGA based performance analysis of stream ciphers ZUC, Snow3G, Grain V1, Mickey V2, Trivium and E0[J]. Microprocessors and Microsystems, 2013, 37(2): 235–245. [DOI: 10.1016/j.micpro.2012.09.007]
[9] WANG L, JING J W, LIU Z B, et al. Evaluating optimized implementations of stream cipher ZUC algorithm on FPGA[C]. In: Information and Communications Security—ICICS 2011. Springer Berlin Heidelberg, 2011: 202–215. [DOI: 10.1007/978-3-642-25243-3_17]
[10] GUAN J, DING L, LIU S K. Guess and determine attack on SNOW3G and ZUC[J]. Journal of Software, 2013, 24(6): 1324–1333. [DOI: 10.3724/SP.J.1001.2013.04287]
关杰, 丁林, 刘树凯. SNOW3G 与 ZUC 流密码的猜测决定攻击 [J]. 软件学报, 2013, 24(6): 1324–1333. [DOI: 10.3724/SP.J.1001.2013.04287]
[11] LIU Z B, ZHANG Q L, MA C Q, et al. HPAZ: A high-throughput pipeline architecture of ZUC in hardware[C]. In: Proceedings of 2016 Design, Automation Test in Europe Conference Exhibition (DATE). IEEE, 2016: 269–272. [DOI: 10.3850/9783981537079_0557]
[12] AVANZI R, BRUMLEY B B. Faster 128-EEA3 and 128-EIA3 software[C]. In: Information Security. Springer Cham, 2015: 199–208. [DOI: 10.1007/978-3-319-27659-5_14]
[13] TRABOULSI S, POHL N, HAUSNER J, et al. Power analysis and optimization of the ZUC stream cipher for LTE-advanced mobile terminals[C]. In: Proceedings of 2012 IEEE 3rd Latin American Symposium on Circuits and Systems (LASCAS). IEEE, 2012: 1–4. [DOI: 10.1109/LASCAS.2012.6180296]
[14] Forward Concepts. Qualcomm leads in global DSP silicon shipments[R/OL]. 2012.
http://www.fwdconcepts.com/dsp111212.htm
[15] Federal Information Processing Standards Publication. NIST FIPS 197: Advanced Encryption Standard (AES)[R/OL]. 2001. [DOI: 10.6028/NIST.FIPS.197]
[16] DAEMEN J, RIJMEN V. The Design of Rijndael: AES—The Advanced Encryption Standard[M]. Springer Berlin Heidelberg, 2013: 1–173. [DOI: 10.1007/978-3-662-04722-4]
[17] GUERON S. Intel® Advanced Encryption Standard (AES) new instructions set Rev. 3.01[R/OL]. Intel Software Network, 2010. https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf
[18] Intel intrinsics guide[R/OL]. INTEL. 2018. https://software:intel:com/sites/landingpage/IntrinsicsGuide/
[19] Advanced vector extensions programming reference[R/OL]. INTEL. 2016.
https: software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf
[20] MATSUDA S, MORIAI S. Lightweight cryptography for the cloud: Exploit the power of bitslice implementation[C]. In: Cryptographic Hardware and Embedded Systems—CHES 2012. Springer Berlin Heidelberg, 2012: 408–425. [DOI: 10.1007/978-3-642-33027-8_24]
[21] BIHAM E. A fast new DES implementation in software[C]. In: Fast Software Encryption—FSE '97. Springer Berlin Heidelberg, 1997: 260–272. [DOI: 10.1007/BFb0052352]
[22] BOGDANOV A, KNUDSEN L R, LEANDER G, et al. PRESENT: An ultra-lightweight block cipher[C]. In: Cryptographic Hardware and Embedded Systems—CHES 2007. Springer Berlin Heidelberg, 2007: 450–466. [DOI: 10.1007/978-3-540-74735-2_31]
[23] SHIBUTANI K, ISOBE T, HIWATARI H, et al. Piccolo: An ultra-lightweight block cipher[C]. In: Cryptographic Hardware and Embedded Systems—CHES 2011. Springer Berlin Heidelberg, 2011: 342–357. [DOI: 10.1007/978-3-642-23951-9_23]

[24] NEVES S, AUMASSON J P. Implementing BLAKE with AVX, AVX2, and XOP[J/OL]. IACR Cryptology ePrint Archive, 2012: 2012/275. https://eprint.iacr.org/2012/275.pdf

[25] AUMASSON J, HENZEN L, MEIER W, et al. SHA-3 proposal BLAKE[R/OL]. Submission to NIST, 2008. (2017-01-20). http://www.131002.net/blake/blake.pdf

[26] 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]

[27] KRAWCZYK H. LFSR-based hashing and authentication[C]. In: Advances in Cryptology—CRYPTO '94, Springer Berlin Heidelberg, 1994: 129–139. [DOI: 10.1007/3-540-48658-5_15]

[28] GUERON S, KOUNAVIS M E. Efficient implementation of the Galois counter mode using a carry-less multiplier and a fast reduction algorithm[J]. Information Processing Letters, 2010, 110(14–15): 549–553. [DOI: 10.1016/j.ipl.2010.04.011]

[29] TAVERNE J, FAZ-HERNÁNDEZ A, ARANHA D F, RODRÍGUEZ-HENRÍQUEZ F, et al. Software implementation of binary elliptic curves: Impact of the carry-less multiplier on scalar multiplication[C]. In: Cryptographic Hardware and Embedded Systems—CHES 2011. Springer Berlin Heidelberg, 2011: 108–123. [DOI: 10.1007/978-3-642-23951-9_8]

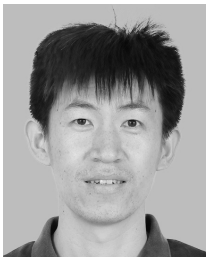
[30] OLIVEIRA T, LPEZ J, ARANHA D F, et al. Lambda coordinates for binary elliptic curves[J/OL]. IACR Cryptology ePrint Archive, 2013: 2013/131. http://eprint.iacr.org//2013/131.pdf

[31] ARANHA D F, FAZ-HERNÁNDEZ A, LÓPEZ J, et al. Faster implementation of scalar multiplication on Koblitz curves[C]. In: Progress in Cryptology—LATINCRYPT 2012, Springer Berlin Heidelberg, 2012: 177–193. [DOI: 10.1007/978-3-642-33481-8_10]

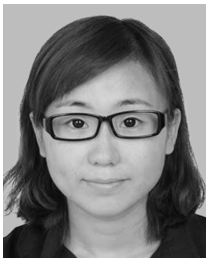
[32] INTEL. Intel® architecture instruction set extensions and future features programming reference[R/OL]. May 2019. http://www.intel.com/design/literature.htm

作者信息

白亮 (1994–), 四川巴中人, 硕士, 工程师. 主要研究领域为密码理论及应用.
gmu.shmily@gmail.com



贾文义 (1982–), 山西吕梁人, 硕士, 高级工程师. 主要研究领域为密码理论及应用.
jiawy163@163.com



朱桂桢 (1987–), 山东滨州人, 博士, 高级工程师. 主要研究领域为密码理论及应用.
zhugz_thu@126.com