课程论文题目

SM4 的快速软件实现技术

摘要

SM4 是中国国家分组密码标准，广泛应用于各种信息系统和安全产品中。一些应用环境对密码算法的软件实现性能有很高的要求。 目前，SM4 软件实现是基于查找表的。 因此，SM4 算法的快速软件实现成为一个重要的研究课题。 这项工作调查了 SM4 的软件优化实现。 使用 SIMD 技术，我们提出了 SM4 的软件优化实现，它比当前基于查找表的软件实现具有显着优势。 在英特尔酷睿 i7-6700 处理器上，与基于查找表的实现相比，软件性能提升了 1.38 倍。

关键词：SM4；SIMD 技术；软件优化实现

1 引言

SM4 算法即原 SMS4 算法，是由国家密码管理局于 2006 年公布的用于无线局域网产品的分组密码算法。SM4 密码算法是国内官方公布的第一个商用密码算法，并于 2012 年 3 月被国家密码管理局批准为行业标准。目前国内应用在安全产品(如 IPSec VPN、SSL、TLS 等)上的密码算法正逐步使用 SM4 算法取代 DES。

市场上的安全产品对使用的密码算法有较高的软件实现性能要求。截至目前，国内外学者对 SM4 软件优化实现方面关注度较小，SM4 的软件实现方法仅限于查表实现。受限于 SM4 算法的设计等因素，SM4 采用查表方法的软件实现性能明显落后于 AES[4]的查表实现。此外，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 加/解密运算的 AES-NI 指令集。采用 AES-NI 指令集实现 AES 算法的软件实现性能是采用查表方法实现 SM4 的 4.23 倍。因此，SM4 算法的快速软件实现技术成为重要研究内容。

近些年，国内外许多学者尝试将采用 SIMD 技术的 SSE/AVX 指令应用到密码算法的软件实现上。SIMD 的全称是“Single Instruction Multiple Data”，即单指令多数据。该技术可实现同一操作并行处理多组数据。SIMD 技术最大的优点在于它的并行性。2012 年，Seiichi 和 shiho 利用 SSE 指令结合 bit-slice 技术应用到 PRESENT、Piccolo，使二者的实现吞吐量分别达到 4.73 和 4.57 cycle/byte。2013 年，Neves 和 Aumasson 利用 AVX2 指令应用到 SHA-3 候选算法 BLAKE 上并提高了其实现性能。但是，截至目前，高端处理器(如 Intel Core i7 等)上的国产密码 SM4 优化实现方面的工作尚没有公开发表。

本文探讨 SM4 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 SM4 的软件优化实现当中。实验结果表明，与目前基于查表的方法相比，SM4 采用 SIMD 技术的软件实现性能有明显优势。

2 相关工作

SM4 是一个分组密码算法，分组长度及主密钥长度均为 128 bit。128 bit 主密钥经密钥编排算法扩展为 32 个 32-bit 轮子密钥，加/解密变换均为 32 轮轮函数 F 的迭代。

2.1 SM4 查表实现

查表实现是密码算法软件实现的最基本方法，该方法是 Daemen 和 Rijmen 在 32-bit 处理器上实现 Rijndael 时提出的。他们将 Rijndael 轮变换操作制成 4 个 8-bit 输入 32-bit 输出的表，一轮 Rijndael 可通过 16 次查表实现。查表方法的核心思想是将密码算法轮函数中尽可能多的变换操作制成表。

3 本文方法

3.1 本文方法概述

SIMD 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。目前，大多数 PC 及服务器采用的是 Intel 处理器。下面就详细阐述使用 SIMD 技术的 AVX2 指令并行实现 SM4 的过程方法。AVX2 指令使用 128-bit xmm 寄存器可实现 4 组消息的并行加/解密，使用 256-bit ymm 寄存器可实现 8 组消息的并行加/解密。接下来首先说明 SM4 并
行实现时的消息存储格式。

3.2 SM4 消息存储格式

将 n 组 128-bit SM4 明文消息记为 Pi (0 ≤ i < n, n=4, 8)。需将 Pi 装载到 4 个 SIMD 寄存器 R0、R1、R2、R3 中。装载规则如下

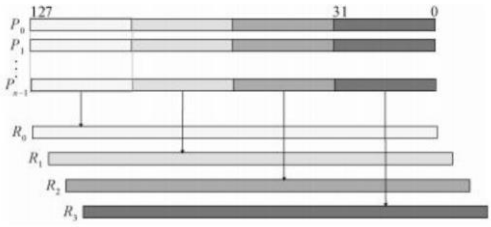

图 1: 消息装载格式

AVX2 指令支持 4/8 道 32-bit 字向量查表操作，因此装载 n 组 SM4 明文消息可通过向量查表指令 vpgatherdd 完成。

3.3 装载轮密钥

SM4 轮函数中包含轮密钥层变换，因此为实现 n 组消息并行加/解密须将 n 个 32-bit 轮密钥装载到 SIMD 寄存器中。将 32 个轮密钥装载到 SIMD 寄存器有 2 种方式：1)每次轮变换时使用 AVX2 指令 vpgatherdd 装载到 SIMD 寄存器中；2)进入 SM4 加/解密操作前，依次将 32 个轮密钥存放在 SIMD 寄存器中。受限于 Intel、AMD 处理器中只有 16 个 SIMD 寄存器，本文采用第 1 种方式。

3.4 轮函数 T 变换

SM4 轮函数中的 T 变换操作是由非线性变换 τ 和线性变换 L 复合而成。并行实现 T 变换有 2 种策略：1)分别实现 τ 和 L；2)将 τ 和 L 合并实现。接下来分别阐述这 2 种策略的实现方法。策略 1：T 变换中的 τ 是由 4 个 S 盒操作并置构成。SM4 所使用的 S 盒规模为 8-bit。不同于轻量级密码所使用的 4-bit S 盒可通过 vpshufb 指令实现。但是，可将 8-bit 的 S 盒输出规模转换为 32-bit，借助 AVX2 的 vpgatherdd 指令即可实现 SM4 S 盒操作。将 8-bit S 盒输出规模转换为 32-bit 有 2 种方法：1)将 8-bit S 盒转换为 4 个 8-bit 输入 32-bit 输出表；2) 8-bit S 盒转换为 2 个 16-bit 输入 32-bit 输出表。策略 2： SM4 中 T 变换包含的非线性变换 τ 与线性变换 L 合并后可制成：1)

4 个 8-bit 输入 32-bit 输出；2) 2 个 16-bit 输入 32-bit 输出的表。如前所述，对于输出规模为 32-bit 的表，可使用 AVX2 中的 vpgatherdd 指令实现并行查表。

3.5 轮函数 F 优化

本节简要介绍 SM4 轮函数 F 实现中的优化技巧。3.2.1 节中使用 4 个 SIMD 寄存器 R0、R1、R2、R3 存储 n 组消息。由图 1 结合式(1)可知，轮函数 F 中 T 变换的输入是 3 个 32-bit 消息与轮密钥异或的结果，该结果需要存储在临时寄存器中。如 3.2.2 节所述，在每轮轮函数变换中均要将轮密钥从内存装载到 SIMD 寄存器中，因此可将存储轮密钥的 SIMD 寄存器 R4 作为该临时寄存器存储 T 变换的输入。此外，由图 1 可知，SM4 的轮函数 F 有如下规律：第 1、2、3 位置的 32-bit 字 R1、R2、R3 和轮密钥 R4 经过 T 变换后异或并更新第 0 位置的 32-bit 字 R0，然后 4 个 32-bit 字循环左移一个 32-bit 位置作为下一轮的输入。循环移位 32-bit 通过调整 SIMD 寄存器的顺序即可，如轮函数 F 的输入为 R0、R1、R2、R3，下一轮的输入为 R1、R2、R3、R0，经过 4 次轮函数变换后寄存器的位置重新为 R0、R1、R2、R3。因此，可以将 SM4 的 4 轮变换展开实现，迭代 8 次即可完成 SM4 加/解密操作。这样实现轮函数 F 既较少使用 SIMD 寄存器的个数，并且消除了不同 SIMD 寄存器内容之间的移动。

4 复现细节

4.1 与已有开源代码对比

```
// 8 组 SM4 消息装载
__m256i id, R0, R1, R2, R3;
vindex=_mm256_setr_epi32
       (0, 4, 8, 12, 16, 20, 24, 28);
R0=_mm256_i32gather_epi32((int*)
    (p + 4 * 0), vindex, 4);
R1=_mm256_i32gather_epi32((int*)
    (p + 4 * 1), vindex, 4);
R2=_mm256_i32gather_epi32((int*)
    (p + 4 * 2), vindex, 4);
R3=_mm256_i32gather_epi32((int*)
    (p + 4 * 3), vindex, 4);
//加载数据
    Temp[0] = _mm256_loadu_si256((const __m256i*)in + 0);
    Temp[1] = _mm256_loadu_si256((const __m256i*)in + 1);
    Temp[2] = _mm256_loadu_si256((const __m256i*)in + 2);
    Temp[3] = _mm256_loadu_si256((const __m256i*)in + 3);


//F 轮优化
```

```
for (int i=0; i < 8; ++i)
{
R4 =_mm256_i32gather_epi32((int*)
   (rk + 4 * i + 0), index, 4);
R4 =_mm256_xor_si256(R4, R1);
R4 =_mm256_xor_si256(R4, R2);
R4 =_mm256_xor_si256(R4, R3);
R4 =T(R4);
R0 =_mm256_xor_si256(R0, R4);
R4 =_mm256_i32gather_epi32((int*)
   (rk + 4 * i + 1), index, 4);
R4=_mm256_xor_si256(R4, R0);
R4 =_mm256_xor_si256(R4, R2);
R4 =_mm256_xor_si256(R4, R3);
R4 =T(R4);
R1 =_mm256_xor_si256(R1, R4);
R4 =_mm256_i32gather_epi32((int*)
   (rk + 4 * i + 2), index, 4);
R4=_mm256_xor_si256(R4, R1);
R4 =_mm256_xor_si256(R4, R0);
R4 =_mm256_xor_si256(R4, R3);
R4 =T(R4);
R2 =_mm256_xor_si256(R2, R4);
R4 =_mm256_i32gather_epi32((int*)
   (rk + 4 * i + 3), index, 4);
R4=_mm256_xor_si256(R4, R2);
R4 =_mm256_xor_si256(R4, R1);
R4 =_mm256_xor_si256(R4, R0);
R4 =T(R4);
R3 =_mm256_xor_si256(R3, R4);
}
// 32 轮迭代（参考 JiaweiJackie 代码）
   for (int i = 0; i < 32; i++) {
      __m256i k =
         _mm256_set1_epi32((enc == 0) ? sm4_key[i] : sm4_key[31 - i]);//装载轮密钥，并判断加密还是解密
      Temp[0] = _mm256_xor_si256(_mm256_xor_si256(X[1], X[2]),
         _mm256_xor_si256(X[3], k));
      //查表
      Temp[1] = _mm256_xor_si256(
         X[0], _mm256_i32gather_epi32((const int*)BOX0,
```

```
            _mm256_and_si256(Temp[0], Mask), 4));
        Temp[0] = _mm256_srli_epi32(Temp[0], 8);
        Temp[1] = _mm256_xor_si256(
          Temp[1], _mm256_i32gather_epi32(
            (const int*)BOX1, _mm256_and_si256(Temp[0], Mask), 4));
        Temp[0] = _mm256_srli_epi32(Temp[0], 8);
        Temp[1] = _mm256_xor_si256(
          Temp[1], _mm256_i32gather_epi32(
            (const int*)BOX2, _mm256_and_si256(Temp[0], Mask), 4));
        Temp[0] = _mm256_srli_epi32(Temp[0], 8);
        Temp[1] = _mm256_xor_si256(
          Temp[1], _mm256_i32gather_epi32(
            (const int*)BOX3, _mm256_and_si256(Temp[0], Mask), 4));

        X[0] = X[1];
        X[1] = X[2];
        X[2] = X[3];
        X[3] = Temp[1];
    }

    //测试部分
    QueryPerformanceCounter(&time_start);//计时开始
      SM4_KeyInit(key, &sm4_key);
      for (int i = 0; i < 2000; i++)
        SM4_Encrypt_x8(in, in, sm4_key);
      QueryPerformanceCounter(&time_over);    //计时结束
```

4.2 创新点

增加了代码的可读性；减少了代码量；改进了数据输入的方式;增加了测试部分。

5 实验结果分析

目前，SM4 采用查表实现方法，实现思想简单，适用于多种平台。查表方法有 2 个明显的缺点：1)CPU 在做查表操作时，由于 SM4 的表规模相对较大，表中的数据在内存和 cache 之间频繁对换导致查表延时较大；2)查表方法无法并行加/解密多组消息，这在一定程度上制约了 SM4 的软件实现性能。相比于查表方法，密码算法使用 SIMD 指令实现时，数据存放在 SIMD 寄存器中，读取数据延时较小，更重要的是 SIMD 指令在并行性方面有明显优势，如轮函数 F 中使用的 vpgatherdd 指令可并行实现查表操作，vpxor(_mm256_xor_si256)指令可并行实现 8 组 SM4 消息 32-bit 字异或操作。此外，对于 4-bit S 盒及按块置换的线性层可使用 vpshufb 指令并行实现。由图 2 可知，相比于查表方法，SIMD 指令实现具有明显的加速效果。

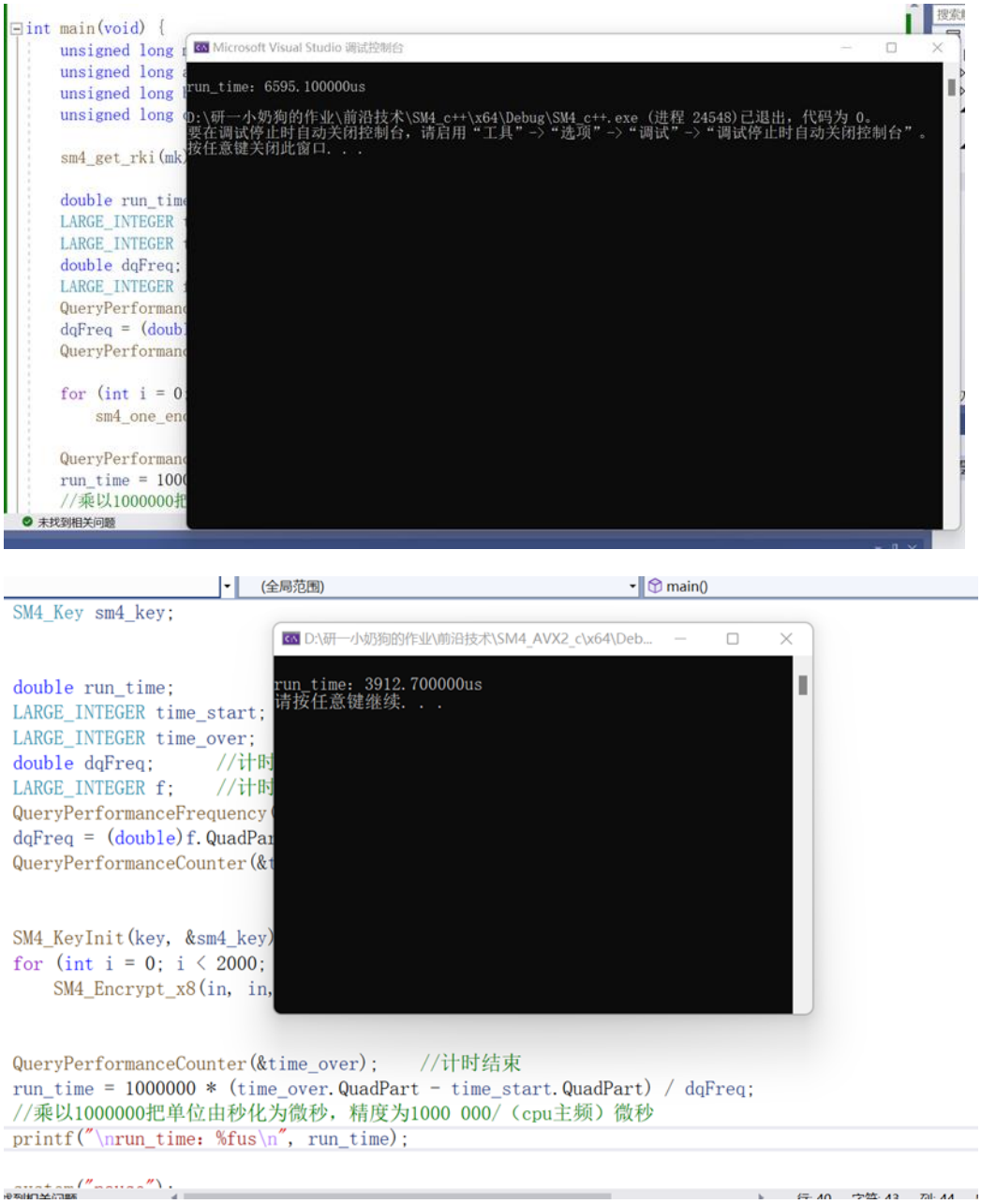

图 2: 实验结果示意

6 总结与展望

本次复现了国密 SM4 算法的软件优化实现方法。使用 vpgatherdd 指令解决了 8-bit S 盒不能使用 SIMD 指令并行实现的问题。利用 SIMD 技术，给出 SM4 轮函数中 T 变换的 2 种软件优化实现策略。实验结果表明，采用 SIMD 技术并行实现 SM4，和目前基于查表的软件实现相比具有明显优势。相比于查表方法，采用 SIMD 技术实现具有明显的加速效果。

参考文献

[1] Lang Huan, Zhang Lei, Wu Wenling. Fast software implementation of SM4. J Univ Chinese Acad Sci, 35 (2) (2018), p. 180.