密码学实验大作业

选题

这次我选的题目是 ZUC 密码算法的优化实现。因为这个题目所涉及到的密码算法相对成熟，而且优化大多是基于硬件实现的优化，所以对软件实现来说，优化空间相对较大。而且，该算法的优化有一定的挑战性。所以我选择了这个题目。

ZUC 密码算法

原理

ZUC 密码算法是一种流密码，每次输出 32 个字节的密钥。密钥的用途在 ZUC 标准中没有指定，但是有其它标准指定了 ZUC 如何在具体应用中使用。ZUC 的一个重要用途，就是在 4G 通信中作为流密码算法使用。而且，由于它的结构相对适合使用硬件实现（这意味着它比较适合嵌入式设备），所以它被选中，成为 4G 通信中的一种基础密码算法。

ZUC 密码算法输入参数是密钥和初始向量。它能够逼近理想的密码，在它的语境中，是完全随机的密钥流。其它的相对核心的部分，包括线性反馈移位寄存器（LFSR）和两个 32 位寄存器 $r1$ 和 $r2$。它通过一系列复杂的位运算和多项式运算，来逼近理想的密钥流。

伪代码

算法 1: S 盒算法

输入: 一个 32 位整数
输出: 一个 32 位整数
数据: 4 个 S 盒替换表
把 32 位整数表示成 4 个字节的形式：$x_0 \| x_1 \| x_2 \| x_3$
for i *= 1 ... 4* **do**
  $y_i \leftarrow$ 第 i 个 S 盒替换表的相应结果
end
return *1*

算法 2: 线性变换 L_1

输入: 32 位无符号整数 x

输出: 32 位无符号整数

return $x \oplus (x <<< 2) \oplus (x <<< 10) \oplus (x <<< 18) \oplus (x <<< 24)$

算法 3: 线性变换 L_2

输入: 32 位无符号整数 x

输出: 32 位无符号整数

return $x \oplus (x <<< 8) \oplus (x <<< 14) \oplus (x <<< 22) \oplus (x <<< 30)$

算法 4: 比特重组

输入: LFSR

输出: 4 个 32 位无符号整数 X_0, X_1, X_2, X_3

$X_0 \leftarrow s_{15\mathrm{H}} \parallel s_{14\mathrm{L}}$

$X_1 \leftarrow s_{11\mathrm{L}} \parallel s_{9\mathrm{H}}$

$X_2 \leftarrow s_{7\mathrm{L}} \parallel s_{5\mathrm{H}}$

$X_3 \leftarrow s_{2\mathrm{L}} \parallel s_{0\mathrm{H}}$

return X_0, X_1, X_2, X_3

算法 5: 非线性函数 F

输入: 3 个 32 位无符号整数 X_0, X_1, X_2

输出: 1 个 32 位无符号整数 W

数据: 2 个 32 位无符号整数 R_1 和 R_2

$W \leftarrow (X_0 \oplus R_1) \boxplus R_2$

$W_1 \leftarrow R_1 \boxplus X_1$

$W_2 \leftarrow R_2 \oplus X_2$

$R_1 \leftarrow S(L_1(W_{1\mathrm{L}} \parallel W_{2\mathrm{H}}))$

$R_2 \leftarrow S(L_2(W_{2\mathrm{L}} \parallel W_{1\mathrm{H}}))$

return W

算法 6: 密钥装入

输入: 128 位密钥 k 和初始向量 iv

输出: 无

数据: LFSR, 各 d_i 构成的表

把 128 位密钥 k 和初始向量 iv 分成 16 个字节，第 i 个字节分别记为 k_i 和 iv_i

$s_i \leftarrow k_i \parallel d_i \parallel iv_i$

算法 7: 初始化模式

输入: LFSR，31 位无符号整数 u

输出: 无

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

$s_{16} \leftarrow (v + u) \mod (2^{31} - 1)$

if $s_{16} = 0$ **then**

  $s_{16} \leftarrow 2^{31} - 1$

end

LFSR $\leftarrow (s_1, s_2, \cdots, s_{16})$

算法 8: 工作模式

输入: LFSR，31 位无符号整数 u

输出: 无

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

$s_{16} \leftarrow v$

if $s_{16} = 0$ **then**

  $s_{16} \leftarrow 2^{31} - 1$

end

LFSR $\leftarrow (s_1, s_2, \cdots, s_{16})$

算法 9: ZUC 密码体制主算法

输入: 128 位的密钥 k, 128 位的初始向量 iv

输出: 密钥字组成的密钥流

数据: LFSR, R_1, R_2

把 k 和 iv 按密钥装入算法装入 LFSR 中 $R_1 \leftarrow 0$ $R_2 \leftarrow 0$

for i = *1 ... 32* **do**

  $X_0, X_1, X_2, X_3 \leftarrow$ 比特重组

  $W \leftrightarrow F(X_0, X_1, X_2)$

  初始化模式 ($W >> 1$)

end

$X_0, X_1, X_2, X_3 \leftarrow$ 比特重组

$W \leftarrow F(X_0, X_1, X_2)$

工作模式

while *true* **do**

  $X_0, X_1, X_2, X_3 \leftarrow$ 比特重组

  $Z \leftarrow F(X_0, X_1, X_2) \oplus X_3$

  输出 32 位密钥字 Z

  工作模式

end

分析

首先，可以比较容易地看出初始化阶段时空复杂度为 $\mathrm{O}(1)$. 然后，由于每次生成 32 位的密钥，都要走一遍工作阶段的步骤，但是使用的空间都是恒定的，所以密钥生成阶段时间复杂度为 $\mathrm{O}(n)$，空间复杂度为 $\mathrm{O}(1)$。其中 n 为要生成的密钥长度。

测试

采用辅助脚本进行自动化测试，辅助脚本文件名为 `zuc_test.py`。脚本能够根据 ZUC 标准文档中的参考 k 和 iv，驱动密码算法生成相应的密钥流，并与参考密钥流相比较，从而验证算法的正确性。运行该文件可以看出，结果正确，这证明算法一般没有问题。

优化的 ZUC 密码算法

原理

一开始编写的 ZUC 密码算法有很多效率较低之处。首先，使用了面向对象的范式来维护状态，这导致对对象的操作开销比较大；其次，数学公式没有经过约简，位运算太多；还有其它各种问题。所以对刚才 ZUC 密码算法，优化的空间还是很大的。

优化尝试

数据结构优化

ZUC 算法中用到的 LFSR，是一个比较明显的优化点。可以看出，算法中对它用到的操作，要么是读取，要么是让它整体位移并且在最后一个元素留出的空位中放入新的元素。因此，可以把它改成一个环形缓冲区，用一个 `int` 类型变量来标记缓冲区的开头。放入元素就相当于在环形缓冲区中放入一个元素，原来的元素会被自动覆盖。

还有一个能优化的点，就是整数位数。由于 Python 中的整数位数是没有限制的，所以可以在多项式计算完毕后，直接模一次。这样能够很好地减少模的次数。

最后一个能优化的点，就是把面向对象去掉。面向对象的写法，是为了维护状态，也就是 LFSR 和那两个寄存器的，如果把它去掉，如何维护状态呢？考虑到密钥生成函数是一个函数，可以在这个函数里定义局部变量，然后利用 Python 中函数可以定义子函数的特性。只要让子函数能够访问父函数中的变量（形成

闭包），就可以做到简单的状态保存，开销非常少，而且增强了可维护性。这样也给后面的操作步骤合并打下了基础。

操作步骤优化

可以看出，无论是初始阶段还是工作阶段，每次的操作步骤大体上是相似的。因此，可以把每轮的操作合并成一个函数。但是，初始阶段不需要返回任何值，工作阶段需要返回，因此需要加入一个参数，把功能区分开。这样，位重组和 F 函数中的位运算，就可以合并起来优化了。

还有一点就是初始化模式能够和工作模式合并。注意到工作模式其实可以用 $u = 0$ 的初始化模式代替，这一点就很显然了。这样能够把精力集中在对多项式算法的优化上，而不是分心分析两个算法。

位运算优化

有了操作步骤优化的基础，就可以进行位运算优化了。首先可以在 F 函数中，把位重组操作和 F 函数本身的操作合并起来：

$$
\begin{aligned}
W &= ((s_{15\mathrm{H}} \parallel s_{14\mathrm{L}}) \oplus R_1) \boxplus R_2 \\
W_1 &= R_1 \boxplus (s_{11\mathrm{L}} \parallel s_{9\mathrm{H}}) \\
W_2 &= R_2 \oplus (s_{7\mathrm{L}} \parallel s_{5\mathrm{H}}) \\
R_1 &= S(L_1(W_{1\mathrm{L}} \parallel W_{2\mathrm{H}})) \\
R_2 &= S(L_2(W_{2\mathrm{L}} \parallel W_{1\mathrm{H}}))
\end{aligned}
$$

然后，就可以对这些位运算进行优化。注意到取 LFSR 中的数时，取高位是取从低到高 31-16 位，但是取 32 位整数时，取高位是取从低到高 32-17 位。同时，以尽量减少和合并运算为原则，进行优化。得到以下结果：

$$
\begin{aligned}
W &= ((s_{15\mathrm{H}} \parallel s_{14\mathrm{L}}) \oplus R_1) \boxplus R_2 \\
W_1 &= R_1 + (s_{11\mathrm{L}} \parallel s_{9\mathrm{H}}) \\
W_{1\mathrm{L}} &= (W_1 \ \&\ \texttt{0x0000ffff}) >> 16 \\
W_{1\mathrm{H}} &= (W_1 \ \&\ \texttt{0xffff0000}) << 16 \\
W_{2\mathrm{L}} &= ((r_2 << 16) \oplus (s_5 << 1)) \ \&\ \texttt{0xffff0000} \\
W_{2\mathrm{H}} &= ((r_2 >> 16) \oplus s_7) \ \&\ \texttt{0x0000ffff} \\
R_1 &= S(L_1(W_{1\mathrm{L}} | W_{2\mathrm{H}})) \\
R_2 &= S(L_2(W_{2\mathrm{L}} | W_{2\mathrm{H}}))
\end{aligned}
$$

其中 $<<$、$|$、$\oplus$ 和 $\&$ 分别表示非循环移位、按位或、按位异或和按位与运算。这样，就能减少位运算的个数，省去不必要的对 W_1 和 W_2 的高位和低位拼接。

另外一个能够优化位运算的地方，是对于 L_1 和 L_2 函数的位运算优化。可以看出，其实不需要对 x 本身做循环移位。由于 Python 的向左和向右移位不考虑符号位，所以它们都是逻辑移位。根据逻辑移位和异或的性质，有以下引理：

$$a \oplus x <<< i \equiv a \oplus (x << i) \oplus (x >> (32 - i))$$

其中 a，x 都是 32 位无符号整数。这样，L_1 和 L_2 函数就可以化简。其实，也可以把 x 向左和向右移位后的结果临时保存，之后再把 x 与 x 向某个方向移位后的结果异或时，异或的位数可以减小。由于 Python 中的移位可能涉及到多个整数的移位，而这种渐进移位的方式之后会移位移得越来越小或者越来越大，所以这种移位方法可以规避一部分没有必要的多个整数移位。

以 L_1 函数为例，原来的表达式为：

$$L_1(x) = x \oplus (x <<< 2) \oplus (x <<< 10) \oplus (x <<< 18) \oplus (x <<< 24)$$

其中 $<<<$ 表示对 32 位无符号整数的向左循环移位运算。现在就可以化简成相应的表达式：

$$\begin{aligned} L_1(x) = x &\oplus (x << 2) \oplus (x >> 8) \\ &\oplus ((x << 2) << 8) \oplus ((x >> 8) >> 6) \\ &\oplus (((x << 2) << 8) << 8) \oplus (((x >> 8) >> 6) >> 8) \\ &\oplus ((((x << 2) << 8) << 8) << 6) \oplus ((((x >> 8) >> 6) >> 8) >> 8) \end{aligned}$$

注意实际上这种嵌套的表示方式意味着用到了两个中间变量。这样，能够减小总共的移位次数，实际上也对不需要桶式移位器（Barrel Shifter）的体系结构有利，虽然在通用计算机上不用考虑这种问题。

多项式运算优化

实际上，模 $2^31 - 1$ 运算可以等到多项式算完结果以后再优化。这是因为根据 ZUC 算法中用到的多项式，哪怕所有的数据都是 $2^31 - 2$，算法的中间结果也不会超出 64 位无符号整数的表示范围。而且，在 Python 中，整数的大小是没有限制的。况且最近的计算机基本上都支持 64 位整数的运算。因此，最后模一次，前面不需要管溢出的问题，这种方法更符合现在的实际。

查表优化

很明显的一个事实是查表是实现 S 盒的比较现实的方法。实现 S 盒的时候，另一个常识是把字节值作为索引。但是，如果体系结构支持 64 位整数，可以把高 32 位用作存放 S 盒结果的临时空间，输出时直接 $>> 32$ 就能得到 S 盒的输出。这时需要把 S 盒的查找表做如下变换：

$$S'_i(b) = S_i(b) << (56 - 8i)$$

这样，原来 S 盒的一个字节，后面就附加上了若干个二进制位 0。这样做的好处是可以直接把 S 盒的结果和需要输出的数字异或起来。例如需要变换的 32 位无符号整数是 0x42424242，那么首先把它扩展成 64 位无符号整数，然后查 S 盒，会得到 4 个 64 位无符号整数：

$$\begin{aligned}
S'_0(\texttt{0x42}) &= \texttt{0xb200000000000000} \\
S'_1(\texttt{0x42}) &= \texttt{0x0011000000000000} \\
S'_2(\texttt{0x42}) &= \texttt{0x0000b20000000000} \\
S'_3(\texttt{0x42}) &= \texttt{0x0000001100000000}
\end{aligned}$$

这时，把这四个 64 位无符号整数分别与原来的 0x0000000042424242 一按位或，就得到了：

$$\begin{gathered}
S'(\texttt{0x42424242}) \\
= S'_0(\texttt{0x42}) \mid S'_1(\texttt{0x42}) \mid S'_2(\texttt{0x42}) \mid S'_3(\texttt{0x42}) \mid \texttt{0x0000000042424242} \\
\texttt{0xb211b21142424242} \\
S'(\texttt{0x42424242}) >> 32 = \texttt{0x00000000b211b211}
\end{gathered}$$

这样，就可以对 S 盒进行优化。不过，由于 Python 中 64 位的整数实际上要用到 3 个底层的 int 来表示（因为 Python 的大整数实现中每“位”是 15 位），所以实际上这样反倒会拖慢效率，其实并没有用这种方法。

线性变换优化

另一种可行的优化 L_1 和 L_2 变换的方法，是把它们转换成 F_2 上的矩阵。矩阵运算的加速，可以通过 numpy 包来实现。不过这种方法最终还是要变成移位运算，因为把运算结果转化为整数时难以避免移位运算。因此，最终没有采用这种方法。

测试

采用自动化脚本进行测试。首先测试优化后的 ZUC 算法的正确性，脚本文件名为 zuc_optimized_test.py，测试原理与未优化的 ZUC 算法类似。类似地，可以看出算法一般是正确的。

然后比较两种 ZUC 算法的性能。脚本文件名为 zuc_bench.py，通过自动化测试框架进行测试。经过测试，可以发现优化后的 ZUC 算法性能大约是未优化的 ZUC 算法的 5 倍，优化效果比较显著。

总结

ZUC 算法的优化

对 ZUC 算法的优化，我这次主要是集中在它在通用计算机上的优化，因为在嵌入式设备中它的优化已经比较成熟了，也有不少论文。在通用计算机上的优化，不是注重硬件方面，更应该注重算法，尤其是数学基础。

系统设计与维护

这次给我的一点启示是系统设计和算法的效率有时候是需要互相妥协的。使用面向对象的范式变成虽然比较优雅，但是可能会降低效率。有的时候，使用场景和系统设计是密不可分的。

对课程的建议

下一次课也不可能有了，所以我想说一下对明年这节课的建议。感觉我们网安学院对系统设计和算法优化强调得确实不大够，所以我希望明年能够多重视这一点，或者这次评分的时候也重视这一点。我自己感觉我的系统设计确实欠缺，算法优化也不是很好。感觉还是得通过密码学实验来提高这两方面的能力。

总结

这次大作业其实非常具有研究性质。我在做大作业的时候，试着去读了一点论文，自己独立研究了算法，也做了一点点推导。感觉这样做能提高我的研究能力。

这个学期的密码学实验就这么结束了。我感觉在密码学实验中，不仅巩固了所学的密码算法，而且提高了编码能力，更锻炼了我的意志力。我感觉这学期的密码学实验课程非常有用。