分组密码算法在 x64 平台上的软件实现速度测试方法研究

季福磊[1,2], 张文涛[1,2], 毛颖颖[3], 赵雪锋[1,2]

[1]中国科学院信息工程研究所 信息安全国家重点实验室 北京 中国 100093
[2]中国科学院大学 网络空间安全学院 北京 中国 100049
[3]国家密码管理局商用密码检测中心 北京 中国 100036

摘要 密码算法的软件实现速度是衡量其实现性能的重要指标之一。在密码算法的设计和评估工作中, 测试密码算法的软件实现速度是一项必不可少的工作。在国内外已有的工作中, 关于如何在 x64 平台上进行密码算法的软件实现速度测试没有形成统一的测试标准。本文以分组密码算法的速度测试为例, 研究如何在 x64 平台上测试密码算法的软件实现速度。首先, 我们通过实验分析在 x64 平台上对密码算法进行软件实现速度测试的过程中容易出现的问题。第二步,我们对目前已有的四种速度测试方法: Matsui 速度测试方法, Fog 速度测试方法, SUPERCOP 速度测试方法和 Gladman 速度测试方法进行研究, 对四种速度测试方法的异同进行比较, 分析四种方法中存在的问题。第三步, 我们采用理论分析与实验探究相结合的研究方法, 研究如何降低速度测试过程中产生的波动性数据对实验结果的影响。我们对速度测试公式选择、样本量选择等问题进行了细致的研究。最终我们给出在 x64 平台上测试分组密码算法软件实现速度的最小值和平均值的有效方法。应用该方法得到的测试结果是稳定的(测试得到的速度随机性小, 结果既不会偏大也不会偏小)、可靠(测试过程取样充分, 测试得到的速度是可信的)、高效的(在保证测试结果可靠和稳定的前提下, 取样量较小, 测试过程耗时较少)。利用本文给出的速度测试方法, 我们对 AES 算法和 SM4 算法在 x64 平台上的软件实现速度进行了实际测试。

关键词 密码算法; x64 平台; 软件实现; 速度测试方法
中图法分类号 TP309 DOI 号 10.19363/J.cnki.cn10-1380/tn.2022.12.05

A Study of Speed Test Method for Implementations of Block Cipher Algorithms on the x64 Platform

JI Fulei[1,2], ZHANG Wentao[1,2], MAO Yingying[3], ZHAO Xuefeng[1,2]

[1]State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China
[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100049, China
[3] State Cryptography Administration Bureau Commercial Cryptography Testing Center, Beijing 100036, China

Abstract The software implementation speed is one of the important criteria of measuring a cryptographic algorithm's performance. In the design and evaluation of a cryptographic algorithm, it is necessary to test the software implementation speed of a cryptographic algorithm. We investigate the research situation at home and abroad, finding that there is no unified test standard on how to test the software implementation speed of cryptographic algorithms on the x64 platform. Taking the speed test of block cipher algorithms as an example, we study how to effectively test the software implementation speed of cryptographic algorithms on the x64 platform. Firstly, we do experiments to analyze the problems that are easy to occur in the process of software speed test on the x64 platform. Secondly, we introduce the existing four speed test methods: Matsui's method, Fog's method, SUPERCOP method and Gladman's method. We compare the similarities and differences of the four speed test methods, and analyze the problems of the four methods. Thirdly, we explore how to reduce the impact of the volatile data on the test results by theoretical analysis and experimental researches. We carefully study and determine the speed test formula and sample size. Finally, we obtain effective methods for testing the minimal and average software implementation speed of block cipher algorithms on the x64 platform. The results obtained by our speed test methods are stable (the randomness of the speed test result is small, and the result is neither too large nor too small), reliable (the speed test process is sufficiently sampled, and the result is worth believing), and efficient (on the premise of ensuring the reliability and stability of the test results, the sample quantity is small and the speed test process takes less time). Applying our new speed test methods, we evaluate the performance of AES and SM4 on the x64 platform.

通讯作者: 张文涛, 博士, 研究员, Email: zhangwentao@iie.ac.cn。
本课题得到国家自然科学基金(No. 61379138)资助。
收稿日期: 2020-06-09; 修改日期: 2020-09-09; 定稿日期: 2022-12-07

Key words cryptographic algorithms; x64 platform; software implementation; speed test method

1 引言

密码算法的软件实现速度是衡量其实现性能的重要指标。在密码算法的设计和评估工作中, 测试密码算法的软件实现速度是一项必不可少的工作。

在 AES (Advanced Encryption Standard)、SHA-3 (Secure Hash Algorithm-3) 等的征集要求[1-2]中, 对算法的三大评估准则为: 安全性 (Security), 计算效率和内存需求 (Cost: computational efficiency and memory requirements) 以及算法及其实现特性 (Algorithm and implementation characteristics)。软件实现速度是衡量算法实现特性的重要因素。

2018 年, 美国国家标准与技术研究院 (National Institute of Standards and Technology, NIST) 发起轻量级密码标准化项目[3] (Lightweight Cryptography Standardization Project), 该项目旨在征集标准化的带关联信息的轻量级认证加密方案 (Lightweight Authenticated Encryption with Associated Data, AEAD) 和散列函数 (Hashing Function)。同年, 中国密码学会举办全国密码算法设计竞赛[4], 竞赛内容包括分组密码算法设计和公钥密码算法设计。在 NIST 标准化征集项目的提交要求和评估标准文件[5]以及中国密码学会算法设计竞赛的技术要求文件[6]中均指出, 密码算法的软件实现速度是重要的衡量指标。

在嵌入式设备和 x64 平台上, 人们采用不同的方法对密码算法的软件实现速度进行测试和评估。

1.1 嵌入式设备上的密码算法软件实现速度测试方法

轻量级密码系统的公平评估平台[7] FELICS (Fair Evaluation of Lightweight Cryptographic Systems) 是由 Daniel 等人开发的开源框架, 它为人们提供了评估轻量级分组密码、流密码在嵌入式设备上的软件实现速度的方法。FELICS的评估指标有三项: 代码量 (代码和数据)、RAM (数据和运行时的栈消耗) 和运行时间。

在运行时间测量方面, 针对不同的测试平台, FELICS 采用不同的方式获得程序运行的时钟周期数:

1) 在 AVR 测试平台上, 使用周期精确模拟器 (cycle accurate simulator) Avrora 来模拟密码操作;

2) 在 MSP 测评平台上, 使用周期精确模拟器 MSPDebug 来模拟密码操作;

3) 在 ARM 测评平台上, 嵌入额外的 C 和汇编代码来读取时钟周期数, 然后在 Arduino Due board 上执行程序。

基于以上操作, FELICS 利用 shell 脚本提取执行测量操作时的时钟周期, 计算系统定时器在测量操作结束时和测量操作开始时的时钟周期之间的差值, 该差值即为运行时间的测量结果[7]。

引文顺序正确, 数量一致。文中最大引文序号为: 23 = 23 (文后文献的最大编号)。

文中施引和文后文献列表概况:
文中施引最末位置 = 6 / 20 页
文中施引的位置数 = 30
文中最大引文序号 = 23
文中实际引用条数 = 23
文后文献最大编号 = 23
文后文献实际条数 = 23。

1.2 x64 平台上的密码算法软件实现速度测试方法

本文对已有的x64平台上的密码算法软件实现速度测试方法进行了调研, 共找到四种方法。它们分别为:

1) Matsui 在论文[8]中提出的速度测试方法;

2) Fog 在其个人网站[9]中给出的速度测试方法;

3) eBACS[10]平台中的 SUPERCOP[11] 测评工具中使用的速度测试方法;

4) Gladman 在其 github 主页上发布的对 AES 进行软件测试的方法[12]中使用的速度测试方法。

在这四种方法中, 速度测试公式、样本量大小、输出参数等各不相同。在下文中, 我们将比较四种方法的异同点, 分析四种方法中存在的问题。

目前, 关于密码算法在 x64 平台上的软件实现速度测试, 没有统一的测试规范。密码算法设计者和评估者往往自行选择测试方法。**本文以分组密码算法为例, 研究密码算法在 x64 平台上的软件实现速度测试方法。通过本文的研究工作, 我们希望得到在 x64 平台上测试密码算法软件实现速度的可靠、稳定、高效的测试方法。**

在探究算法软件实现速度测试的有效方法时, 本文参照以下三个标准:

1) 测试方法得到的结果是**可靠的**, 即测试过程取样充分, 测试得到的速度是可信的;

2) 测试方法得到的结果是**稳定的**, 即测试得到的速度随机性小, 结果既不会偏大也不会偏小;

3) 测试方法得到的结果是**高效的**, 即在保证测试结果可靠和稳定的前提下, 取样量较小, 测试过程耗时较少。

1.3 本文的主要贡献

1) 对现有的四种 x64 平台上的分组密码算法[①]软件实现速度测试方法: Matsui 方法、Fog 方法、SUPERCOP 方法、Gladman 方法进行介绍。对四种测试方法的速度计算公式、样本量选择、方差控制等方面的异同点进行比较。

2) 探究在 x64 平台上进行算法软件实现速度测试的过程中普遍存在的问题, 分析四种速度测试方法中存在的问题。

密码算法软件实现速度测试中存在的普遍性问题为: 单次实现速度激增, 平均速度的测试结果不稳定且偏大。

Matsui 方法、Fog 方法、SUPERCOP 方法中存在的问题为: 计算均值 (或中位数) 时不考虑标准差导致结果随机性较大且数值偏大。

Gladman 方法中存在的问题为: 最小速度测试方法容易导致结果偏大, 最小值与均值测试函数分离且取样数不一致导致测试结果不稳定。

3) 对速度测试的计算公式选择、样本量大小选择、排除波动性数据等问题进行了实验研究和理论分析。提出用于测试 x64 平台上密码算法软件实现速度最小值的方法-方法 6。提出用于测试 x64 平台上密码算法软件实现速度最小值、平均值和标准差的方法-方法 7。

方法 6 和方法 7 可用于测试任意密码算法在 x64 平台上的软件实现速度, 包括但不只局限于分组密码算法、公钥密码算法、流密码、杂凑函数和认证加密方案等; 在方法 6 和方法 7 中, 我们将待测试的密码算法函数 (记为 foo) 设计为独立的输入模块; 在对不同的密码算法 (例如杂凑函数 Keccak[13]、认证加密方案 ASCON[14]、分组密码算法 AES[15]、GIFT[16] 等) 的软件实现速度进行测试时, 只需将 foo 替换为要测试的密码算法即可。

4) 对方法 6 和方法 7 的可行性进行分析。将方法 6 和方法 7 的测试结果与 Matsui 方法、Fog 方法、SUPERCOP 方法、Gladman 方法的测试结果进行比较。经过可行性分析与实验结果比较, 得出结论: 利用方法 6 和方法 7 得到的测试结果是稳定的、可靠的、高效的。

5) 应用方法 6 和方法 7, 测试 AES 算法和 SM4 算法在 x64 平台上的软件实现速度。

2 准备工作

2.1 符号说明

S[·]	S 盒操作。
<<<	左循环移位操作。
$GF(2^8)$	2^8 上的有限域。
$\oplus$	按位异或操作。
Z_2^n	n 比特的 0–1 向量集合。
cpb	处理单位字节长度的消息所消耗的时钟周期数 (cycles per byte)。
foo	待测试的算法函数, 包括加密算法、解密算法等。
$x\cdot$foo	连续执行 foo , 共执行 x 次。
loops	循环执行次数。
len	foo 函数处理的消息长度, *len* 的单位为字节。
readtsc()	读时钟周期函数, 该函数的返回值为 __rdtsc() [17]。
cycles(P)	执行程序块 P 所需的时钟周期数。
cy[*i*]	第 i 次执行 $x\cdot$foo 得到的测试速度。
t_0	空转计时的时间消耗。
$v_{\min}$	最小速度。
av	均值。
sig	标准差。
$quar_1, quar_3$	第一四分位数, 第三四分位数。
med	中位数。
B，KB，MB	字节(Bytes), 千字节(KBytes), 兆字节(MBytes)。

2.2 声明

本文中对 AES 和 SM4 算法进行的所有速度测试实验均符合以下声明:

1) 采用 ECB 模式[18]对不同长度的消息进行加解密速度测试, 测试的消息长度均是密码算法分组长度的倍数。

在方法 6 和方法 7 中, 待测试的密码算法函数 (记为 foo) 是独立的输入模块。在测试过程中, 操作

① 为使叙述简洁, 在不做专门说明时, 下文中提到的“密码算法”均指代“分组密码算法”。

模式 (例如 CTR 模式、CBC 模式等) 和消息长度等信息是 foo 的输入参数。当消息长度短于密码算法的分组长度、或最后一个消息分组需要填充、或操作模式发生变化时, 这些具体操作均在 foo 中进行, 方法 6 和方法 7 的测试过程保持不变。

2) 所有实验均在作者的个人台式机电脑上进行。电脑参数为: Intel(R) Core(TM) i7-6700 3.40GHz CPU, 16GB RAM, 64-位 Windows 操作系统。

实验所用电脑的一级指令缓存为 4×32KB, 一级数据缓存为 4×32KB, 二级缓存为 4×256KB, 三级缓存为 8MB。

所有程序均采用C语言编程, 采用Visual Studio 2010 Intel C/C++ Complier 编译, 编译环境为 Release x64。

3) 对 AES 和 SM4 算法的软件实现均为单线程的基本实现, 未进行多线程的优化实现。所有实验中测试的加解密函数中均不包含密钥扩展步骤。

4) 通过计算密码算法对单位长度消息进行加解密所需要的时钟周期数 (cycles per byte, cpb) 来衡量目标密码算法的加解密软件实现速度。该值越小, 表示密码算法处理单位长度消息所需要的时钟周期数越少, 即实现速度越快。

5) 在程序运行过程中, 调用读时钟周期函数 readtsc() 也会产生时钟周期的消耗。因此, 我们在计算函数消耗的时钟周期之前, 首先计算空转计时的时间消耗。以第 4 节中的方法 4.1 为例, 计算空转的时间消耗的方法为:

$$t_0 = \text{(double)readtsc( )}$$

$$t_0 = \text{(double)readtsc( )} - t_0$$

为使文章叙述简洁, 在下文的所有测试函数中, 我们不再详细表述空转计时时间消耗的计算过程, 而是直接使用 t_0 表示空转计时的时间消耗。

6) 为降低测试的随机性, 我们在测试之前关闭超线程 (Intel Hyper-Threading), 关闭 CPU 超频加速 (Intel Turbo Boost), 将进程与 CPU 绑定, 关闭其他不必要的进程。

7) 为防止 CPU 乱序执行 (Out-of-order Execution) 对速度测试的结果产生影响, 在调用 readtsc( ) 函数和执行 $x\cdot$foo 之前和之后, 我们执行以下汇编代码, 实现 CPU 序列化:

```
xor   eax, eax

cpuid
```

2.3 算法简介

2.3.1 AES 算法

高级加密标准[15](Advanced Encryption Standard, AES)是由 Daemen 和 Rijmen 设计的, 原名为 Rijdael[19] 的分组密码算法。2001 年, AES 由 NIST 发布于 FIPS PUB 197[15]。此后, AES 成为对称密钥加密中最流行的算法之一。

AES 共有三个版本, 其中消息分组长度均为 128 比特, 密钥长度分别为 128 比特、192 比特和 256 比特, 对应的加解密轮数分别为 10 轮、12 轮和 14 轮。我们将 AES 的三个版本分别记为 AES-128、AES-192 和 AES-256。

AES 为 SPN 结构的分组密码算法, 它的轮函数由四个步骤构成, 分别为: 字节替换 (SubBytes)、行移位 (ShiftRows)、列混淆 (MixColumns) 和轮子密钥加 (AddRoundKey)。

记 AES 的 128 比特输入为 $b_0b_1\cdots b_{127}$, 将其表示为 16 个字节形式的输入 $B_0B_1\cdots B_{15}$, 其中 $B_i=b_{8i}b_{8i+1}\cdots b_{8i+7}$。将 $B_i\,(0\leqslant i\leqslant 15)$ 表示为矩阵形式(称为 AES 的状态矩阵):

$$\begin{pmatrix} B_0 & B_4 & B_8 & B_{12} \\ B_1 & B_5 & B_9 & B_{13} \\ B_2 & B_6 & B_{10} & B_{14} \\ B_3 & B_7 & B_{11} & B_{15} \end{pmatrix}$$

记加密轮数为 $n\,(n=10$, 12 或 14)。记初始密钥为 $k_0k_1\cdots k_{15}$ (对应 AES-128) 或 $k_0k_1\cdots k_{23}$ (对应 AES-192) 或 $k_0k_1\cdots k_{31}$ (对应 AES-256), 记第 $j\,(0\leqslant j\leqslant n)$ 轮的轮子密钥为 $k_0^jk_1^j\cdots k_{15}^j$, 其中 $k_i,k_i^j\in Z_2^8$。记 128 比特输出为 $c_0c_1\cdots c_{127}$。

则 AES 的加密过程可表示为算法 1:

算法 1 AES 加密过程

Algorithm 1 The encryption process of AES

AES 加密过程	
	Input: $b_0b_1\cdots b_{127}$, $k_0k_1\cdots k_{15}$ (or $k_0k_1\cdots k_{23}$, or $k_0k_1\cdots k_{31}$)
	Output: $c_0c_1\cdots c_{127}$
1	AddRoundKey
2	for $i=0$ to $n-1$ do
3	SubBytes
4	ShiftRows
5	MixColumns
6	AddRoundKey
7	end for

8	SubBytes
9	ShiftRows
10	AddRoundKey

AES 加密过程中的四个操作步骤如下:

1) 字节替换操作 SubBytes; 对所有的 B_i $(0 \leqslant i \leqslant 15)$, 计算 $B_i = \mathrm{S}[B_i]$;

2) 行移位操作 ShiftRows; 将状态矩阵中的第 i $(0 \leqslant i \leqslant 3)$ 行左循环移位 $8i$ 比特, 即,

$$B_i B_{i+4} B_{i+8} B_{i+12} = (B_i B_{i+4} B_{i+8} B_{i+12}) <<< 8i$$

3) 列混淆操作 MixColumns; 将状态矩阵中的第 i $(0 \leqslant i \leqslant 3)$ 列进行 $\mathrm{GF}(2^8)$ 上的乘法运算, 左乘矩阵 $\boldsymbol{M}$, 即,

$$\begin{pmatrix} B_{4i} \\ B_{4i+1} \\ B_{4i+2} \\ B_{4i+3} \end{pmatrix} = \boldsymbol{M} * \begin{pmatrix} B_{4i} \\ B_{4i+1} \\ B_{4i+2} \\ B_{4i+3} \end{pmatrix}$$

其中 $\boldsymbol{M}$ 是 $\mathrm{GF}(2^8)$ 上的矩阵:

$$\boldsymbol{M} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix}$$

$\mathrm{GF}(2^8)$ 上使用的不可约多项式为:

$$x^8 + x^4 + x^3 + x + 1$$

4) 轮子密钥加操作 AddRoundKey; 在加密开始时, 首先计算 $B_i = B_i \oplus k_i$, $0 \leqslant i \leqslant 15$; 在第 j $(0 \leqslant j \leqslant n)$ 轮, 计算 $B_i = B_i \oplus k_i^j$, $0 \leqslant i \leqslant 15$。

AES 的解密过程与加密过程结构相同, 在解密过程中: 使用 S 盒的逆操作 $\mathrm{S}^{-1}[\cdot]$、$\boldsymbol{M}$ 矩阵的逆矩阵 $\boldsymbol{M}^{-1}$; 改左循环移位为右循环移位; 逆序使用加密密钥。关于 AES 的 S 盒取值、密钥编排算法、解密过程等的更多细节, 请参阅文献[19]。

本文采用构造表格[20]的方式对 AES 算法进行软件实现, 每个字节的 SubBytes、ShiftRows 和 MixColumns 操作可通过一步查表实现。

2.3.2 SM4 算法

SM4 算法[21]是由国家商用密码管理办公室公布的一种分组密码算法。SM4 算法的分组长度为 128 比特, 密钥长度为 128 比特, 加解密轮数为 32 轮。

SM4 算法采用非线性迭代结构, 以 32 比特为单位进行加解密运算, 称一次迭代运算为一轮变换。设 SM4 的 128 比特输入为 $(X_0, X_1, X_2, X_3) \in (Z_2^{32})^4$, 128 比特输出为 $(Y_0, Y_1, Y_2, Y_3) \in (Z_2^{32})^4$, 轮密钥为 $rk_i \in Z_2^{32}$, $i = 0,1,\cdots,31$, 则轮函数 F 可表示为:

$$\mathrm{F}(X_0, X_1, X_2, X_3, rk_i) = X_0 \oplus \mathrm{T}(X_1 \oplus X_2 \oplus X_3 \oplus rk_i)$$

定义反序变换函数 R:

$$\mathrm{R}(A_0, A_1, A_2, A_3) = (A_3, A_2, A_1, A_0)$$

则 SM4 的加密过程可表示为算法 2:

算法 2 SM4 加密过程

Algorithm 2 The encryption process of SM4

SM4 加密过程	
	Input: (X_0, X_1, X_2, X_3); rk_i, $i = 0,1,\cdots,31$
	Output: (Y_0, Y_1, Y_2, Y_3)
1	for $i = 0$ to 31 do
2	$X_{i+4} = \mathrm{F}(X_i, X_{i+1}, X_{i+2}, X_{i+3}, rk_i)$
3	end for
4	$(Y_0, Y_1, Y_2, Y_3) = \mathrm{R}(X_{32}, X_{33}, X_{34}, X_{35})$

在轮函数 F 中, $\mathrm{T}: Z_2^{32} \to Z_2^{32}$ 是一个可逆变换, 由非线性变换 τ 和线性变换 L 符合而成: $\mathrm{T}(\cdot) = \mathrm{L}(\tau(\cdot))$。

1) 非线性变换 τ; τ 由 4 个并行的 S 盒构成; 设 τ 的输入值为 $A = (a_0, a_1, a_2, a_3) \in (Z_2^8)^4$, 输出值为 $B = (b_0, b_1, b_2, b_3) \in (Z_2^8)^4$, 则有:

$$(b_0, b_1, b_2, b_3) = \tau(A) = (\mathrm{S}[a_0], \mathrm{S}[a_1], \mathrm{S}[a_2], \mathrm{S}[a_3])$$

2) 线性变换 L; 设 L 的输入值为 $B \in Z_2^{32}$, 输出值为 $C \in Z_2^{32}$, 则有:

$$C = \mathrm{L}(B) =$$

$$B \oplus (B <<< 2) \oplus (B <<< 10) \oplus (B <<< 18) \oplus (B <<< 24)$$

SM4 算法的解密过程与加密过程结构相同, 解密过程中逆序使用加密密钥。关于 SM4 的 S 盒取值、密钥编排算法、解密过程等的更多细节, 请参阅文献[22]。

本文采用构造表格[21]的方式对 SM4 算法进行软件实现。

3 现有的密码算法软件实现速度测试方法介绍

本节对现有的四种密码算法软件实现速度测试

方法进行介绍: Matsui 速度方法, Fog 速度测试方法, SUPERCOP 速度测试方法和 Gladman 速度测试方法。并对四种速度测试方法的各项特征进行总结和比较。

3.1 Matsui 速度测试方法

在文章[8]中, Matsui 等人研究分组密码算法在奔腾III和奔腾 4 处理器(Intel Pentium III and 4 Processors)上的优化实现方案。

在该论文中, 作者给出了对密码算法进行软件实现速度测试的方法: 连续测试运行单次 foo 所消耗的时钟周期数, 对多次的测试结果取平均值, 作为最终的测试结果。

具体测试方法为:

1) 依次执行 foo , 循环执行 *loops* 次; 记录每次的测试结果:

$$cy[i]=(\text{cycles(foo)}-t_0)/len$$

2) 计算 $cy[i]\,(0\leqslant i\leqslant loops-1)$ 数组的平均值, 即: $av=(cy[0]+cy[1]+\cdots+cy[loops-1])/loops$ 该测试过程的伪代码如方法 1 所示。

方法 1 Matsui 速度测试方法

Method 1 Matsui's method of testing the average speed

Matsui 速度测试方法	
	Input: foo,*loops*,*len*
	Output: *av*
1	Procedure Timing
2	for $i=0$ to $loops-1$ do
3	$cy[i]=\text{readtsc()}$
4	foo
5	$cy[i]=(\text{readtsc()}-cy[i]-t_0)/len$
6	end for
7	$av=(cy[0]+cy[1]+\cdots+cy[loops-1])/loops$

3.2 Fog 速度测试方法

Fog 在其个人网站中给出了一系列关于软件优化实现的资源, 其中包括了分析代码在 Windows 和 Linux 系统上运行速度的测试程序。

利用该测试程序, 使用者可以测量代码运行消耗的时钟周期和性能监视器计数器(performance monitor counters), 如缓存丢失, 分支预测错误, 资源停滞等。

在速度测试方面, 作者连续测试运行单次 foo 所消耗的时钟周期数, 考虑到单次测试结果的不稳定性, 作者对所有的测试结果进行输出。

具体测试方法为:

1) 依次执行 foo , 循环执行 *loops* 次; 记录每次的测试结果:

$$cy[i]=(\text{cycles(foo)}-t_0)/len$$

2) 对所有的 $cy[i]\,(0\leqslant i\leqslant loops-1)$ 进行输出。该测试过程的伪代码如方法 2 所示。

方法 2 Fog 速度测试方法

Method 2 Fog's method of testing the speed

Fog 速度测试方法	
	Input: foo,*loops*,*len*
	Output: $cy[i]\,(0\leqslant i\leqslant loops-1)$
1	Procedure Timing
2	for $i=0$ to $loops-1$ do
3	$cy[i]=\text{readtsc()}$
4	foo
5	$cy[i]=(\text{readtsc()}-cy[i]-t_0)/len$
6	end for

3.3 SUPERCOP 速度测试方法

密码系统的 ECRYPT II 基准测试系统[22-23]是由欧洲委员会第六框架计划 (the European Commission's Sixth Framework Programme, FP6) 资助的项目。它主要测试分组密码和杂凑函数在 Atmel AVR 8-bit RISC 微处理器上的实现性能。

eBACS[10] (ECRYPT Benchmarking of Cryptographic Systems) 是由 ECRYPT II 扩展而来的测评框架。它包括四个部分, 分别用于测试非对称密码、流密码、杂凑函数和认证加密方案的性能。eBACS 平台使用其开发的 SUPERCOP[11] 工具完成测评工作。

在速度测试方面, SUPERCOP 记录每次运行 foo 所消耗的时钟周期数, 并对所有的测试结果进行输出。考虑到单次运算的速度具有不稳定性, SUPE-RCOP计算所有测试数据的中位数和四分位数。

具体测试方法为:

1) 依次执行 foo , 循环执行 *loops* 次; 记录每次执行 foo 前的时钟周期值:

$$cy[i]=\text{readtsc()}$$

2) 计算执行第 $i\,(0\leqslant i\leqslant loops-1)$ 次 foo 所消耗的时钟周期数:

$$cy[i]=(cy[i+1]-cy[i])/len$$

3) 计算 $cy[i]\,(0\leqslant i\leqslant loops-1)$ 数组的中位数

med 和四分位数 $quar_1, quar_3$。

该测试过程的伪代码如方法 3 所示。

方法 3 SUPERCOP 速度测试方法

Method 3 SUPERCOP method of testing the average speed

SUPERCOP 速度测试方法

Input: foo, $loops$, len

Output: med, $quar_1$, $quar_3$

```
1  Procedure Timing
2    for i = 0 to loops − 1 do
3      cy[i] = readtsc()
4      foo
5    end for
6    for i = 0 to loops − 1 do
7      cy[i] = (cy[i + 1] − cy[i]) / len
8  end for
9 Compute med, quar1, quar3 of cy[i] (0 ≤ i ≤ loops − 1)
```

3.4 Gladman 速度测试方法

Gladman 在其 github 主页发布了对 AES 进行软件测试的方法。在速度测试部分, 该方法测试运行速度的最小值、平均值和标准差。

3.4.1 时钟周期最小值测试方法

测试多次运行 foo 所需的时钟周期数的最小值。其中, foo 为加密函数、解密函数或密钥扩展函数。

具体测试方法为:

1) 执行一次 foo, 去除一次运算的不稳定性。

2) 依次执行 foo 和 $(x+1)\cdot$foo, 循环执行 $loops$ 次; 当循环次数 i 大于 $loops$ 的 10%后, 计算

$$cy_1 = \min_{loops} \text{cycles(foo)}$$

和

$$cy_2 = \min_{loops} \text{cycles}((x+1)\cdot \text{foo})$$

然后计算 $et = (cy_2 - cy_1)/x$。

3) 计算 $v_{\min} = et/len$。该测试过程的伪代码如方法 4 所示。

方法 4 Gladman 最小速度测试方法

Method 4 Gladman's method of testing the minimal speed

Gladman 最小速度测试方法

Input: foo, x, $loops$, len, $c1 = \text{INF}$, $c2 = \text{INF}$

Output: $v_{\min}$

```
1   Procedure Timing
2     foo
3     for i = 0 to loops − 1 do
4       cy1 = readtsc()
5       foo
6       cy1 = readtsc() − cy1
7       cy2 = readtsc()
8       (x + 1)·foo
9       cy2 = readtsc() − cy2
10      if i > (loops/10) then
11        c1 = (c1 < cy1 ? c1 : cy1)
12        c2 = (c2 < cy2 ? c2 : cy2)
13      end if
14    end for
15    et = (c2 − c1)/x
16    v_min = et/len
17  end Timing
```

(注: $c2$ 和 $c1$ 的值均包含调用读时钟周期函数产生的时钟周期消耗, 因此两数相减可以消除 t_0 对实验结果的影响。)

在 AES 速度测试中, 作者设置 $loops$=100, x=8。

3.4.2 时钟周期平均值和标准差测试方法

测试多次运行 $y\cdot$foo 所需的时钟周期数的平均值和标准差。

具体测试方法为:

1) 对执行 $y\cdot$foo 所需的时钟周期个数进行初始采样 SAMPLE1 和正式采样 SAMPLE2。

2) 执行初始采样 SAMPLE1。循环执行 $loops_1$ 次 $y\cdot$foo, 计算 SAMPLE1 中 cycles($y\cdot$foo) 的均值 av_1 和标准差 sig_1, 规定:

$$sig_1 \geqslant 0.05 \times av_1$$

(上式是为了保证 SAMPLE1 中得到的 cycles($y\cdot$foo) 的标准差的值不会过小。)

3) 执行正式采样 SAMPLE2。循环执行 $loops_2$ 次 $y\cdot$foo。在 $loops_2$ 次的采样过程中, 当某次采样的时间(记为 cy)在 $[av_1 - sig_1, av_1 + sig_1]$ 范围内时, 将这次采样记为一次有效采样。计算 SAMPLE2 中 cycles($y\cdot$foo) 的所有有效采样的均值 av_2 和标准差 sig_2。

4) SAMPLE2 结束后, 只有当有效采样次数(记为 $loops_{good}$)满足:

$$loops_{good} \geqslant 90\% \cdot loops_2$$

并且 SAMPLE2 标准差 sig_2 满足:

$$sig_2 < 10\% \cdot av_2$$

时, 才认为时间测试成功, 返回 TRUE。若测试不成功, 重新采样测量。

5) 若连续测试 10 次都没有成功, 则将标准差的可接受限度放低 5%, 再次重新采样测量。若当标准差的可接受限度降到了均值的 30% 时, 仍没有测试成功, 则返回 FALSE, 测试失败。

该测试过程的伪代码如方法 5 所示。

方法 5 Gladman 速度均值、标准差测试方法

Method 5 Gladman's method of testing the average speed and the standard deviation of the speed

Gladman 平均速度、速度标准差测试方法

Input: $\text{foo}, y, loops_1, loops_2, len$

$loops_{good} = 0, tol = 0.1, lcnt = 0$

Output: av_2, sig_2

```
1   Procedure SAMPLE1
2     for i = 1 to loops_1 do
3     cy = readtsc()
4     y · foo
5     cy = readtsc() − cy
6     av_1 += cy, sig_1 += cy * cy
7     end for
8     av_1 = av_1 / loops_1
9   sig_1 = sqrt((sig_1 − av_1 * av_1 * loops_1) / loops_1)
10    sig_1 = (sig_1 < 0.05 * av_1 ? 0.05 * av_1 : sig_1)
11    call SAMPLE2
12  end SAMPLE1

13  Procedure SAMPLE2
14    for i = 1 to loops_2 do
15    cy = readtsc()
16    y · foo
17    cy = readtsc() − cy
18    if cy > (av_1 − sig_1) and cy < (av_1 + sig_1)
      then
19      av_2 += cy, sig_2 += cy * cy
20      loops_good ++
21    end if
22    end for
23    if loops_good > 0.9 · loops_2 then
24      av_2 = av_2 / loops_good
25      sig_2 = sqrt((sig_2 − av_2 * av_2 * loops_good) /
        loops_good)
26    if sig_2 > tol · av_2 then
27      loops_good = 0, restart SAMPLE1
28    end if
29    else
30    if lcnt++ == 10 then
31      lcnt = 0, tol = tol + 0.05
32      if tol > 0.3 then return FAUSE
33      loops_good = 0, restart SAMPLE1
34    end if
35    end if
36  end SAMPLE2
37  sig_2 = sig_2 / av_2
38  av_2 = (av_2 − t_0) / (y * len)
39  return TRUE
```

在 AES 速度测试中, 作者设置 $loops_1$= 1000、$loops_2$= 10000、y =16。

3.5 四种速度测试方法比较

为对四种速度测试方法进行直观比较, 我们对四种测试方法的特征进行总结, 记录到表 1 中。

根据对四种测试方法的介绍, 结合表 1, 我们可以发现, 四种方法的不同主要体现在两个方面:

1) 速度计算公式不同, 用来衡量测试结果的标准不同。

i. 观察方法 1 和方法 2,可以发现, Matsui 方法和 Fog 方法中采用的速度计算公式是相同的, 只是最终的输出结果不同。Fog 方法对所有 $cy[i]$ ($0 \leqslant i \leqslant loops-1$)的值进行输出; Matsui 方法输出 $cy[i]$ ($0 \leqslant i \leqslant loops-1$)的平均值 av。

ii. **在平均速度的衡量方面。**SUPERCOP 方法选择计算 $cy[i]$ ($0 \leqslant i \leqslant loops-1$)的中位数 med; Matsui 方法和 Gladman 方法选择计算平均值 av。Matsui 方法对 av 的标准差 sig 不做限制; Gladman 方法要求 sig 必须控制在 30%以内。

iii. **在最小速度的衡量方面。**Gladman 方法采用的最小速度计算公式为: $v_{\min}$ = (min cycles $((x+1)\cdot \text{foo}) - \min \text{cycles}(\text{foo}))/(x*len)$。根据 Matsui 方法 (或 Fog 方法), 我们可以考虑利用公式: $v_{\min} = (\min(x \cdot \text{foo}))/(x*len)$ 计算最小速度。在第 5 节中, 我们通过实验对以上两个公式的测试效果进行比较。

表 1 四种速度测试方法总结

Table 1 The summary of the four speed test methods

测试方法	计算公式	连续执行 foo 次数 (x)	循环执行次数 ($loops$)	输出结果	是否控制标准差大小
Fog 方法	$\text{cycles}(x\cdot\text{foo})$	1	无说明	$cy[i], 0\leqslant i\leqslant loops-1$	否
Gladman 方法	$\min\text{cycles}((x+1)\cdot\text{foo})-\min\text{cycles}(\text{foo})$	8	100	$v_{\min}$	否
	$\text{cycles}(x\cdot\text{foo})$	16	10000	av, sig	是 $sig\leqslant 30\%$
Matsui 方法	$\text{cycles}(x\cdot\text{foo})$	1	无说明	av	否
SUPERCOP 方法	$\text{cycles}(x\cdot\text{foo})$	1	无说明	$med, quar_1, quar_3$	否

表 2 方法 6 和方法 7 总结

Table 2 The summary of method 6 and method 7

测试方法	计算公式	连续执行 foo 次数 (x)	循环执行次数 ($loops$)	输出结果	是否控制标准差大小
方法 6	$\min\text{cycles}(x\cdot\text{foo})$	20	1000	$v_{\min}$	否
方法 7	$\text{cycles}(x\cdot\text{foo})$	20	1000	$v_{\min}, av, sig$	是 $sig\leqslant 30\%$

2) **连续执行 foo 的次数 x 的取值不同。**

Matsui 方法、Fog 方法和 SUPERCOP 方法中, $x=1$。Gladman 方法中, 测试最小速度时 $x=8$, 测试平均速度时 $x=16$。

4 现有的密码算法软件实现速度测试方法中存在的问题

我们首先在 4.1 节中探究密码算法软件实现速度测试中存在的普遍性问题。在此基础上, 我们在 4.2 节中结合实验数据分析 Matsui 方法(即 Fog 方法)、SUPERCOP 方法中存在的问题, 在 4.3 节中分析 Gladman 方法中存在的问题。

4.1 密码算法软件实现速度测试中存在的普遍性问题

当我们对一个密码函数 foo 进行速度测试时, 最直观的方法是: 记录执行 $x\cdot$foo 所需的时间, 然后计算最小速度 $v_{\min}$ 和平均速度 av。如下文方法 4.1 所示, 是对函数 foo 进行连续 $loops$ 次速度测试的直观方法。

方法 4.1 对函数 foo 进行速度测试的直观方法

Method 4.1 The intuitive method of testing the speed of function foo

对函数进行速度测试的直观方法

Input: foo, x, $loops$, len

Output: $cy[i]$ $(0\leqslant i\leqslant loops)$, $v_{\min}$, av

```
Procedure Timing
    for i = 0 to loops − 1 do
    cy[i] = readtsc()
    x · foo
    cy[i] = (readtsc() − cy[i] − t0)/(x · len)
    end for
    v_min = min_i {cy[i]}
    av = (cy[0] + cy[1] + ··· + cy[loops − 1])/loops
    end Timing
```

本节以 AES-128 为例, 说明在 x64 平台对密码算法进行软件速度测试的过程中存在的普遍性问题。

4.1.1 问题一: 单次实现速度激增

实验 1

采用方法 4.1, 测试 AES-128 加密消息长度为 2048B 的数据的速度, 记录每次加密所需的平均时钟周期数。

实验参数为: foo 表示 AES-128 加密函数; $x=1$; $loops=1000$; $len=2048$B。

我们重复上述实验 12 次, 其中共有 7 次的实验结果中出现单次测试速度超过 50.0 cpb 的情况。选取其中三次的实验结果, 记录在表 3 中。

速度测试的 cpb 值越大, 表示算法处理单位长度消息所消耗的时钟周期数越多, 即算法的实现效率越低。由表 3 可以看出, 在连续 1000 次测试中, 有 990 次左右的测试结果保持在 13.00～11.30 cpb 的范

表 3 实验 1 测试结果

Table 3 The Results of Experiment 1

编号 i	加密速度 $cy[i]$ (cpb)	编号 i	加密速度 $cy[i]$ (cpb)	编号 i	加密速度 $cy[i]$ (cpb)
642	109.17	845	74.87	60	69.23
909	19.11	941	21.84	868	21.30
699	16.71	529	18.53	945	20.10
0	15.36	214	13.70	3	19.73
804	15.21	109	12.86	207	19.21
75	14.93	633	12.49	101	19.06
390	14.17	0	12.48	7	16.83
495	13.17	950	12.47	16	16.21
180	13.13	947	12.31	941	14.01
490	12.48	974	12.27	837	13.16
其他	12.30～11.34	其他	12.25～11.33	其他	12.93～11.30

围内, 有八次左右的测试结果超过 16.00 cpb, 甚至会出现测试结果大于 100.00 cpb 的情况。

实验 1 的结果显示, 在对算法进行软件实现速度测试时, 会出现单次实现速度激增的情况。在 2.2 节中我们已经声明, 我们采取多种措施降低测试的随机性, 例如关闭超线程、关闭 CPU 超频加速、将进程与 CPU 绑定、关闭其他不必要的进程、设置 CPU 序列化等。但仍然存在一些不可避免的随机性因素, 导致测试结果不稳定。因此, 在速度测试的取样过程中, 我们应该设置判断条件, 排除单次实现速度激增对测试结果产生的影响。

4.1.2 问题二: 平均速度的测试结果不稳定, 且偏大

根据方法 4.1, 我们可以分别测试算法运算速度的最小值和平均值。为研究 x 的取值对最小值和平均值测试结果的影响, 我们进行以下实验:

实验 2

采用方法 4.1, 测试 AES-128 加密消息长度为 1024B 和 2048B 的数据的速度, 记录每次加密所需的最小时钟周期数和平均时钟周期数。

实验参数为: foo 表示 AES-128 加密函数; $loops = 1000$; len = 1024B、2048B; x 的取值为 1～30。

我们将实验结果整理在表 4 中。

表 4 实验 2 测试结果

Table 4 Results of Experiment 2

x	消息长度: 1024B						消息长度: 2048B					
	第一次测试		第二次测试		第三次测试		第一次测试		第二次测试		第三次测试	
	Min (cpb)	*Ave* (cpb)	*Min* (cpb)	*Ave* (cpb)	*Min* (cpb)	*Ave* (cpb)	*Min* (cpb)	*Ave* (cpb)	*Min* (cpb)	*Ave* (cpb)	*Min* (cpb)	*Ave* (cpb)
1	11.13	12.41	11.14	11.71	11.13	11.64	11.00	11.40	11.01	11.44	11.02	11.40
5	11.25	11.96	11.24	11.83	11.25	11.64	11.09	11.44	11.08	11.6	11.10	11.63
8	11.24	11.95	11.24	11.95	11.24	11.63	11.10	12.06	11.09	11.71	11.10	11.61
9	11.26	12.00	11.25	11.94	11.26	11.67	11.10	11.72	11.09	11.74	11.09	11.48
10	11.26	12.16	11.26	11.81	11.19	11.68	11.10	11.94	11.09	11.62	11.09	11.50
15	11.27	11.91	11.27	11.86	11.26	11.63	11.10	11.95	11.10	11.66	11.10	11.44
16	11.27	11.86	11.27	11.83	11.27	11.62	11.10	11.77	11.10	11.64	11.10	11.50
18	11.27	12.26	11.27	11.87	11.27	11.65	11.10	11.92	11.10	11.76	11.10	11.51
20	11.27	12.23	11.27	11.84	11.27	11.65	11.10	11.80	11.10	11.84	11.10	11.48
25	11.27	12.01	11.27	11.88	11.27	11.67	11.11	11.81	11.10	11.72	11.10	11.49
30	11.27	12.01	11.27	11.82	11.27	11.90	11.10	11.81	11.11	11.68	11.10	11.50

(注: “*Min*”列中的数据表示最小速度 $v_{\min}$, “*Ave*”列中的数据表示平均速度 av。)

由表 4 可以看出, 在取样量充分的情况下 (实验 2 中, $loops = 1000$):

1) 最小速度的测试结果是比较稳定的, 但是平均速度的测试结果波动较大;

2) 对比最小速度和平均速度的值, 发现平均速度的测试结果偏大;

3) x 的取值对最小速度的测试效果没有明显影响。

平均速度测试结果不可靠的原因: 在 4.1.1 节中我们发现, 在密码算法的软件实现过程中, 存在单次实现速度激增的情况, 这一现象导致了平均速度的测试结果的不稳定, 且容易导致平均值的测试结果偏大。

在对算法进行软件实现速度测试时, 若我们需要测试平均速度, 应该排除单次实现速度激增对测试结果的影响, 一方面要保证取样量充分, 另一方面可以通过控制测试数据的标准差等途径排除波动较大的数据。

4.2 Matsui 方法(即 Fog 方法)和 SUPERCOP 方法中存在的问题

我们通过实验直观地观察 Matsui 方法(即 Fog 方法)和 SUPERCOP 方法的测试效果。为更好地说明问题, 我们将这三种方法的测试结果与 Gladman 方法的测试结果也进行对比。

我们进行如下实验:

实验 3

1) 利用 **Matsui 方法**(即 Fog 方法), 计算加解密函数的平均速度: foo 为 AES-128 加密函数(或解密函数); $x=1$; $loops=1000$; len 的取值为 128B 到 4096B。

2) 利用 **SUPERCOP 方法**, 计算加解密函数速度数组 $cy[i]$ ($0\leqslant i\leqslant loops-1$) 的中间值: foo 为 AES-128 加密函数(或解密函数); $x=1$; $loops=1000$; len 的取值为 128B 到 4096B。

3) 利用 **Gladman 方法中的方法 4**, 计算加解密函数的最小速度: foo 为 AES-128 加密函数(或解密函数); $x=8$; $loops=100$; len 的取值为 128B 到 4096B。

4) 利用 **Gladman 方法中的方法 5**, 计算加解密函数的平均速度和统计标准差: foo 为 AES-128 加密函数(或解密函数); $y=16$; $loops_1=1000$, $loops_2=10000$; len 的取值为 128B 到 4096B。

我们将实验 3 得到的测试结果整理在表 5 中。

表 5 实验 3 测试结果

Table 5 Results of experiment 3

	Enc				Dec			
	Matsui 方法 (Fog 方法)	SUPERCOP 方法	Gladman 方法		Matsui 方法 (Fog 方法)	SUPERCOP 方法	Gladman 方法	
len (Bytes)	*Ave* (cpb)	*Med* (cpb)	*Min* (cpb)	*Ave* (cpb)	*Ave* (cpb)	*Med* (cpb)	*Min* (cpb)	*Ave* (cpb)
128	13.96	15.63	13.71	13.66(1.25%)	14.20	15.81	13.95	13.86(1.02%)
256	12.88	13.41	12.47	12.32(0.71%)	12.76	13.59	12.64	12.53(0.84%)
384	12.68	12.69	12.05	11.90(0.70%)	12.37	12.85	12.25	12.11(0.74%)
512	12.32	12.37	11.94	11.78(0.86%)	12.14	12.52	12.11	11.95(0.72%)
640	12.09	12.13	11.80	11.66(0.76%)	12.01	12.30	11.97	11.83(0.60%)
768	11.80	11.98	11.72	11.55(0.75%)	11.98	12.14	11.89	11.75(0.77%)
896	12.12	11.87	11.66	11.48(0.77%)	12.04	12.04	11.83	11.67(0.73%)
1024	11.64	11.79	11.61	11.43(0.69%)	12.98	11.96	11.77	11.61(0.61%)
2048	11.67	11.50	11.44	11.27(0.79%)	11.96	11.71	11.63	11.45(0.69%)
3072	11.60	11.46	11.41	11.22(0.80%)	11.69	11.61	11.55	11.42(0.93%)
4096	11.69	11.38	11.38	11.22(0.91%)	11.65	11.55	11.52	11.41(0.89%)

(注: “*Med*”列中的数据表示所有 $cy[i]$ ($0\leqslant i\leqslant loops-1$)的中位数 *med*, “*Min*”列中的数据表示最小速度 $v_{\min}$, “*Ave*”列中的数据表示平均速度 *av*, “*Ave*”列中括号中的百分数表示统计测试的标准差 *sig*。)

观察表 5, 我们发现:

1) Matsui 方法(即 Fog 方法)和 SUPERCOP 方法得到的测试结果随机性较大, 测试结果不稳定, 测试结果的数值整体偏大;

2) Gladman 方法在计算平均速度时, 控制测试数据的平均差小于 30%, 通过 Gladman 方法得到的平均速度的数值是比较稳定的, 而且不会出现测试结果偏大的情况;

3) 在 Gladman 方法的测试结果中, 会出现最小速度的值大于平均速度的值的情况, 导致最小值与平均值失去可比性;

我们将 Matsui 方法(即 Fog 方法)和 SUPERCOP 方法中存在的问题总结为以下一点。

计算均值 (或中位数) 时不考虑标准差导致结果随机性较大且数值偏大。

导致这一问题的原因为: Matsui 方法(即 Fog 方法)和 SUPERCOP 方法中, 均采用多次测试 $x\cdot$foo ($x=1$) 的速度的方式来衡量 foo 的软件实现速度, 在计算均值 (或中位数) 时, 没有考虑数据的标准差, 因此无法排除不稳定的测试数据对实验结果的影响。在 5.1 节中我们已经指出, 直接计算测试数据的平均值, 得到的结果是不稳定的, 且数值会偏大。

4.3 Gladman 方法中存在的问题

为直观地展现 Gladman 测试方法中存在的问题, 我们运行 Gladman 测试方法, 测试 AES-128 加密函

数和解密函数的软件实现速度。实验参数设置如下:

实验 4

1) 利用**方法 4**, 计算加解密函数的最小速度: foo 为 AES-128 加密函数(或解密函数); $x = 8$; $loops = 100$; *len* 的取值为 128B 到 4096B;

2) 利用**方法 5**, 计算加解密函数的平均速度和统计标准差: foo 为 AES-128 加密函数(或解密函数); $y = 16$; $loops_1 = 1000$, $loops_2 = 10000$; *len* 的取值为 128B 到 4096B。

我们将实验 4 得到的测试结果整理在表 6 中。

表 6 实验 4 测试结果

Table 6 Results of experiment 4

len (Bytes)	Enc		Dec	
	Min (cpb)	*Ave* (cpb)	*Min* (cpb)	*Ave* (cpb)
128	13.71	13.66(1.25%)	13.95	13.86(1.02%)
256	12.47	12.32(0.71%)	12.64	12.53(0.84%)
384	12.05	11.90(0.70%)	12.25	12.11(0.74%)
512	11.94	11.78(0.86%)	12.11	11.95(0.72%)
640	11.80	11.66(0.76%)	11.97	11.83(0.60%)
768	11.72	11.55(0.75%)	11.89	11.75(0.77%)
896	11.66	11.48(0.77%)	11.83	11.67(0.73%)
1024	11.61	11.43(0.69%)	11.77	11.61(0.61%)
2048	11.44	11.27(0.79%)	11.63	11.45(0.69%)
3072	11.41	11.22(0.80%)	11.55	11.42(0.93%)
4096	11.38	11.22(0.91%)	11.52	11.41(0.89%)

(注: "*Min*"列中的数据表示最小速度 $v_{\min}$, "*Ave*"列中的数据表示平均速度 *av*, "*Ave*"列中括号中的百分数表示统计测试的标准差 *sig*。)

观察表 6 可以看出, 利用 Gladman 方法测试得到的结果有以下特点:

1) 6.1 节中表 12 所示的结果是利用本文在 5.2 节中最终给出的方法 6 测试 AES-128 加解密短消息速度的测试结果; 对比表 6 和表 12 的实验结果可以看出, 利用 Gladman 方法测试得到的加解密最小速度的值偏大;

2) 平均值的测试结果比较稳定, 平均值的标准差维持在百分之十以内;

3) 容易出现最小值大于平均值的情况, 最小值与平均值的结果不具有可比性; 当 *len* 的取值为 128B 到 4096B 时, Enc 和 Dec 的速度测试结果都出现了最小值大于或等于平均值的情况;

4) 最小值与平均值的测试函数分开, 取样数和测试样本不同, 也导致最小值与平均值的结果可比性降低。

我们将 Gladman 速度测试方法中存在的问题总结为以下两点。

4.3.1 最小速度测试方法容易导致结果偏大

Gladman 速度测试方法中, 速度最小值的计算公式为:

$$v_{\min} = (\min\text{cycles}(9 \cdot \text{foo}) - \min\text{cycles}(\text{foo}))/(8 * len)$$

为使叙述更加简洁, 在下文中, 我们将最小速度的计算公式简记为:

$$v_{\min} = (\min(9 \cdot \text{foo}) - \min(\text{foo}))/8$$

采用上式测试最小速度, 会导致最终得到的结果偏大。这是因为: 若测试过程中 min(foo) 的结果较小, 那么 $(\min(9 \cdot \text{foo}) - \min(\text{foo}))/8$ 的计算结果会偏大。我们在第 7.2 节中, 通过实验直观地说明公式 $v_{\min} = (\min((x+1) \cdot \text{foo}) - \min(\text{foo}))/x$ 的测试效果。

4.3.2 最小值与均值测试函数分离, 且取样数不一致, 导致测试结果不稳定

在方法 5 所示的 Gladman 均值标准差测试函数中, 为保证最终得到的均值和标准差数值稳定, 在正式采样 SAMPLE2 中, 作者设置 $loops_2 = 10000$。当我们测试的函数加解密的消息长度较大 (大于 1MB) 时, $loops_2 = 10000$ 将导致速度计算函数运行速度较慢, 无法快速得到结果。

因此, Gladman 将测试速度最小值和平均值的函数分开。在消息长度较小时, 分别使用方法 4 和方法 5 测试最小速度和平均速度; 在消息长度较大时, 只测试最小速度, 不测试平均速度和标准差。

以上设置存在两个问题:

1) **最小值与平均值测试函数取样数不一致, 最小值的取样数和连续运算次数偏小**; 方法 4 中

loops= 100 , x = 8; 方法 5 中 *loops*= 10000 , y = 16; 最小值的取样数 *loops* 和连续运算次数 x 较小, 导致最小值的测试结果偏大;

2) **最小值的测试函数与平均值的测试函数分离, 测试样本不一致**; 测试函数分离的情况下, 计算最小值和平均值的测试样本是两个不同的样本, 导致最小值的测试结果与平均值的测试结果不具有可比性。

5 本文的方法

在本节中, 我们通过实验探究测试密码算法软件实现速度最小值和平均值的可靠、稳定、高效的方法。

本节仍以测试 AES-128 加密 2048B 长度消息的速度为例。本节最终给出测试密码算法加解密长消息长度最小速度的方法-方法 6, 给出测试密码算法加解密短消息长度最小速度、平均速度和标准差的方法-方法 7。

5.1 测试函数的适用环境

方法 7 可以同时计算算法软件实现速度的最小值、平均值和标准差, 它的最小取样数为 $1000\times 20\times 2$; 方法 6 可以计算软件实现速度的最小值, 它的取样数为 1000×20 。

当待测试的消息长度较短时, 我们建议使用方法 7; 若测试的消息长度过长, 导致方法 7 运算时间较长时, 我们可以使用方法 6 测试运算速度的最小值。以 AES 为例, 当待测试消息长度小于 1MB 时, 建议使用方法 7 测试加解密速度; 当待测试消息长度大于 1MB 时, 建议使用方法 6 测试加解密速度。

5.2 探究最小速度测试方法

5.2.1 探究 *x* 的取值

根据 4.1.2 节中实验 2 的结果, 我们已经得到结论: x 的取值对最小速度的测试效果没有明显影响。

考虑到单次速度测试的结果可能存在不稳定性(单次实现速度激增等), **本文最终选择 $x=20$** 。

5.2.2 探究最小速度的计算公式

1) 首先考虑 Gladman 方法中的计算公式, 即利用:

$$v_{\min}=(\min((x+1)\cdot \mathrm{foo})-\min(\mathrm{foo}))/x \tag{1}$$

计算最小速度。

2) 作为尝试, 我们考虑利用下式计算最小速度:

$$v_{\min}=(\min((x+1)\cdot \mathrm{foo}-\mathrm{foo}))/x \tag{2}$$

在 4.1.1 节中我们已经说明, 单次运算存在速度突然增加的问题。采用 (2) 式计算最小速度, 若某次 $1\cdot$foo 的测试结果很大, 将导致最终的测试结果偏小, 甚至会出现负值的情况。因此我们不选用公式 (2)。

3) 考虑 Matsui 方法(或 Fog 方法)中的测试函数, 即利用下式计算最小速度:

$$v_{\min}=(\min(x\cdot \mathrm{foo}))/x \tag{3}$$

为比较公式 (1) 和公式 (3) 的测试效果, 我们进行如下实验:

实验 5

1) 采用公式 (1), 测试加密函数的最小速度: foo 为 AES-128 加密函数; $x=20$; *loops* = 1000 ; *len* 的取值为 128～4096B;

2) 采用公式 (3), 测试加密函数的最小速度: foo 为 AES-128 加密函数; $x=20$; *loops* = 1000 ; *len* 的取值为 128～4096B;

我们将实验结果整理在表 7 中。为使实验结果更有说服性, 我们重复多次执行实验 5, 最终在表 7 中记录三次的实验结果。

分析表 7 的数据, 比较使用公式 (1) 和公式 (3) 得到的测试结果, 可以发现:

1) 使用公式 (1) 测试得到的结果整体偏大; 这是因为, 若 min(foo) 的测试结果比较小, 那么 $(\min((x+1)\cdot \mathrm{foo})-\min(\mathrm{foo}))/x$ 的值就会偏大;

2) 使用公式 (3) 得到的测试结果较为稳定。

综合以上分析, **本文最终采用公式 (3) 作为最小速度的计算公式**。

5.2.3 探究 *loops* 的取值

我们需要合理选择 *loops* 的值。一方面 *loops* 的值应足够大, 这样做可以有效排除单次运算速度激增对测试结果的影响, 提高测试结果的稳定性; 另一方面 *loops* 的值不宜过大, 这样做可以提高速度测试的效率。

为确定 *loops* 的取值, 我们进行实验 6, 借以观察 *loops* 取值对结果的影响。

实验 6

采用公式 (3) 进行速度测试。foo 表示 AES-128 加密函数; x = 20; *len* = 2048B; *loops* 的取值为 50～10000。

我们将实验结果整理在表 8 中。

由表 8 可以看出, 当 *loops* 的值大于 700 时, 测试结果是比较稳定的。**为保证测试结果的稳定性, 同时保证测试函数的效率, 本文最终选择 *loops* = 1000** 。

表 7 实验 5 测试结果

Table 7 Results of experiment 5

len (Bytes)	公式(1)测试结果			公式(3)测试结果		
	第一次测试 *Min* (cpb)	第二次测试 *Min* (cpb)	第三次测试 *Min* (cpb)	第一次测试 *Min* (cpb)	第二次测试 *Min* (cpb)	第三次测试 *Min* (cpb)
128	13.78	13.76	13.76	13.59	13.55	13.60
256	12.53	12.53	12.52	12.40	12.42	12.40
384	12.12	12.12	12.19	12.03	12.03	12.05
512	11.99	11.99	12.05	11.91	11.92	11.92
640	11.84	11.85	11.90	11.78	11.79	11.79
768	11.76	11.76	11.79	11.69	11.72	11.70
896	11.69	11.69	11.72	11.69	11.69	11.70
1024	11.64	11.64	11.66	11.64	11.64	11.65
2048	11.47	11.47	11.48	11.45	11.45	11.45
3072	11.43	11.43	11.43	11.39	11.39	11.39
4096	11.39	11.39	11.39	11.35	11.35	11.35

(注: “*Min*”列中的数据表示最小速度 $v_{\min}$。)

表 8 实验 6 测试结果

Table 8 Results of experiment 6

loops	最小速度 $v_{\min}$ (cpb)		
	第一次测试	第二次测试	第三次测试
50	11.28	11.29	11.29
100	11.31	11.29	11.28
300	11.28	11.29	11.28
400	11.28	11.28	11.28
500	11.29	11.28	11.28
700	11.28	11.28	11.28
900	11.29	11.28	11.29
1000	11.29	11.28	11.28
3000	11.28	11.28	11.28
5000	11.28	11.28	11.27
7000	11.28	11.28	11.28
9000	11.27	11.28	11.28
10000	11.27	11.28	11.28

根据以上研究, 我们最终确定测试最小速度方法, 记该方法为**测试函数一**。具体测试过程为:

1) 执行一次 foo, 去除一次运算的不稳定性。

2) 执行 $x\cdot\text{foo}$, 循环执行 *loops* 次, 其中 $x=20$, $loops=100$; 当循环次数 i 大于 *loops* 的 10%后, 计算:

$$c1=\min_{loops}\text{cycles}(x\cdot\text{foo})$$

3) 计算 $et=(c1-t_0)/x$; 其中 t_0 为空转计时的时间消耗。

4) 计算 $v_{\min}=et/len$。

测试函数一的伪代码如方法 6 所示:

方法 6 测试函数一: 本文给出的测试最小速度的方法

Method 6 Testing function one: the method of testing the minimal speed proposed by us

测试函数一

```
    Input: foo, x = 20, loops = 1000, len, c1 = INF
    Output: v_min
1   Procedure TimingMin
2       foo
3       for i = 1 to loops do
7         cy = readtsc()
8         x · foo
9         cy = readtsc() − cy
10        if i > (loops/10) then
11            c1 = (c1 < cy ? c1 : cy)
13        end if
14      end for
15      et = (c1 − t0)/x
16      v_min = et/len
17  end Timing
```

5.3 优化平均速度和标准差测试方法

在平均速度的测试方面, Gladman 方法通过控制标准差等措施, 排除了波动性较大的数据对实验结果的影响, 是一种比较可靠的测试方法。但该方法仍然存在一些问题, 例如计算最小值与平均值的函数分离, 选用的样本量和测试数不一致, 导致两个结果可比性降低; 选取的样本数较大, 导致测试效率较低等。

本文采用 Gladman 方法的测试框架测试平均速

度和标准差。为提高测试结果的可靠性和可比性, 提高测试的效率, 我们对 Gladman 方法做以下两点改进。

5.3.1 探究 $loops_2$ 的取值

如方法 5 所示, 在计算速度的平均值和标准差时, 首先进行初始采样 SAMPLE1, 得到平均值的一个可接受的范围:

$$(av_1 - sig_1, av_1 + sig_1) \tag{4}$$

在正式采样 SAMPLE2 中, 每次采样的时间 cy 只有在式 (4) 的范围内, 才被记为一次有效采样。

在 SAMPLE1 中, 取样数 $loops_1 = 1000$。在 SAMPLE2 中, 取样数 $loops_2 = 10000$。

为提高测试算法的运算效率, 我们通过实验观察 $loops_2$ 的取值对实验结果的影响。

实验 7

利用方法 5 进行速度测试。foo 表示 AES-128 加密函数; x = 20; len = 2048B; $loops_2$ 的取值为 50～10000。

我们将实验结果整理在表 9 中。

表 9 实验 7 测试结果

Table 9 Results of experiment 7

$loops_2$	平均速度 av_2 (cpb)		
	第一次测试	第二次测试	第三次测试
50	11.45 (2.52%)	11.43 (1.11%)	11.39 (0.43%)
100	11.42 (1.33%)	11.45 (0.96%)	11.49 (0.69%)
300	11.45 (0.65%)	11.42 (0.61%)	11.39 (0.28%)
400	11.52 (2.51%)	11.49 (1.39%)	11.43 (0.86%)
500	11.48 (2.48%)	11.46 (1.45%)	11.57 (6.93%)
700	11.44 (0.87%)	11.43 (0.71%)	11.40 (0.51%)
900	11.44 (2.50%)	11.44 (0.79%)	11.46 (0.85%)
1000	11.43 (0.62%)	11.44 (0.74%)	11.43 (1.01%)
3000	11.43 (0.62%)	11.44 (0.77%)	11.43 (0.68%)
5000	11.42 (0.59%)	11.43 (0.72%)	11.42 (1.24%)
7000	11.43 (0.72%)	11.41 (0.51%)	11.43 (0.66%)
9000	11.43 (0.64%)	11.44 (1.50%)	11.43 (0.66%)
10000	11.42 (0.65%)	11.43 (0.64%)	11.42 (0.67%)

(注: 括号中的百分数表示统计测试的标准差 sig。)

观察表 9 的实验结果可以发现, 当 $loops_2$ 的值大于 900 时, 利用方法 5 计算得到的平均值的结果保持稳定。**为保证测试结果的稳定性, 同时提高测试效率, 本文最终选择 $loops_2 = 1000$。**

5.3.2 将计算最小值和平均值的函数整合为一个函数

我们将计算最小值和平均值的函数整合为同一个函数。在同一个函数中, 我们可以利用相同的采样数和采样样本计算最小值和平均值, 利用该种方式测试得到的最小值和平均值具有更高的可比性。

根据以上研究, 我们得到同时测试密码算法软件实现最小速度、平均速度和标准差的方法, 记为**测试函数二**, 具体测试方法为:

1) 对执行 $y \cdot$foo 所需的时钟周期个数进行初始采样 SAMPLE1 和正式采样 SAMPLE2;

2) 执行初始采样 SAMPLE1; 循环执行 $loops_1$ 次 $y \cdot$foo, 其中 $loops_1 = 1000$, $y = 20$; 计算 SAMPLE1 中 cycles($y \cdot$foo) 的均值 av_1 和标准差 sig_1, 规定:

$$sig_1 \geqslant 0.05 \times av_1$$

上式是为了保证 SAMPLE1 中得到的 cycles($y \cdot$foo) 的标准差的值不会过小;

3) 执行正式采样 SAMPLE2; 循环执行 $loops_2$ 次 $y \cdot$foo, 其中 $loops_2 = 1000$, $y = 20$; 在 $loops_2$ 次的采样过程中, 当某次采样的时间(记为 cy)在 $[av_1 - sig_1, av_1 + sig_1]$ 范围内时, 将这次采样记为一次有效采样; 计算 SAMPLE2 中 cycles($y \cdot$foo) 的所有有效采样的最小值 $v_{\min}$、均值 av_2 和标准差 sig_2;

4) SAMPLE2 结束后, 只有当有效采样次数 $loops_{good}$ 满足:

$$loops_{good} \geqslant 90\% \cdot loops_2$$

并且 SAMPLE2 标准差 sig_2 满足:

$$sig_2 < 10\% \cdot av_2$$

时, 才认为时间测试成功, 返回 TRUE。若测试不成功, 重新采样测量;

5) 若连续测试 10 次都没有成功则将标准差的可接受限度放低 5%, 再次重新采样测量。若当标准差的可接受限度降到了均值的 30% 时, 仍没有测试成功, 则返回 FALSE, 测试失败。

测试函数二的伪代码如方法 7 所示:

方法 7 测试函数二: 本文给出的同时测试最小速度、平均速度和标准差的方法

Method 7 Testing function two: the method of testing the minimal speed, the average speed and the standard deviation proposed by us

测试函数二

Input:

$$\text{foo}, y = 20, loops_1 = 1000, loops_2 = 1000, len$$

$$loops_{good} = 0, tol = 0.1, lcnt = 0$$

```
Output: v_min, av_2, sig_2
Procedure SAMPLE1
    for i = 1 to loops_1 do
    cy = readtsc()
    y · foo
    cy = readtsc() − cy
    av_1 += cy, sig_1 += cy * cy
    end for
    av_1 = av_1 / loops_1
sig_1 = sqrt((sig_1 − av_1 * av_1 * loops_1) / loops_1)
    sig_1 = (sig_1 < 0.05 * av_1 ? 0.05 * av_1 : sig_1)
    call SAMPLE2
end SAMPLE1

Procedure SAMPLE2
    c1 = INF
    for i = 1 to loops_2 do
    cy = readtsc()
    y · foo
    cy = readtsc() − cy
    if cy > (av_1 − sig_1) and cy < (av_1 + sig_1) then
        av_2 += cy, sig_2 += cy * cy
        loops_good ++
        c1 = (c1 < cy ? c1 : cy)
    end if
    end for
    if loops_good > 0.9 · loops_2 then
    av_2 = av_2 / loops_good
    sig_2 = sqrt((sig_2 − av_2 * av_2 * loops_good) / loops_good)
    if sig_2 > tol · av_2 then
        loops_good = 0, restart SAMPLE1
    end if
    else
    if lcnt++ == 10 then
        lcnt = 0, tol = tol + 0.05
        if tol > 0.3 then return FAUSE
        loops_good = 0, restart SAMPLE1
    end if
    end if
end SAMPLE2
    v_min = (c1 − t_0v_min) / (x * len)
    sig_2 = sig_2 / av_2
    av_2 = (av_2 − t_0) / (x * len)
    return TRUE
```

(注: $t_{0v_{\min}}$ 为在计算最小速度的过程中, 调用读时钟周期函数产生的时钟周期消耗。)

5.4 可行性分析与实验验证

5.4.1 可行性分析

在测试密码算法的软件实现速度时, 出现波动性数据不可避免。本文旨在排除波动性数据对测试结果的影响, 得到稳定、可靠、高效的速度测试方法。本文采取理论分析与实验探究相结合的研究方式。

下面我们对方法 6 和方法 7 的可行性进行分析。

1) 测试最小速度的方法-方法 6。

i. **速度计算公式的选择**。我们首先通过理论分析排除选择公式 (2), 然后通过实验观察公式 (1) (3) 的测试结果。实验 5 的结果显示, 公式 (3) 的测试结果稳定。

ii. **测试样本量的选择**。样本量的取值必须通过实验探究得到。样本量较大可以帮助测试者排除波动性数据。同时, 考虑到测试效率, 样本量不宜过大。我们分别通过实验 2 和实验 6 选取了 x 和 $loops$ 的取值: x=20, $loops$=1000。

速度测试的效率与具体密码算法的实现速度有关。本文重点考虑分组密码算法, 它们的实现速度相差不大, 因此可以共用 x 和 $loops$ 的取值。

2) 同时测试最小速度和平均速度的方法-方法 7。

i. **速度计算公式的选择**。沿用公式(3)计算最小速度。选择公式:

$$av = (\sum_{loops_2} y \cdot \text{foo}) / (y * loops_2) \tag{5}$$

计算平均速度。

ii. **测试样本量的选择**。我们通过实验探索样本量大小对测试结果的影响。测试样本量的大小应在保证测试结果稳定性和可靠性的同时, 提高测试效率。我们分别通过实验 2 和实验 7 选取了 y 和 $loops_2$ 的取值: y=20, $loops_2$=1000。

速度测试的效率与具体密码算法的实现速度有关。本文重点考虑分组密码算法, 它们的实现速度相差不大, 因此可以共用 y 和 $loops_2$ 的取值。

iii. 我们使用同一组测试数据同时计算最小速度和平均速度, 使得二者的测试结果具备可比性。

iv. 在测试过程中, 我们限制数据的标准差不超过 30%。该方法可以有效减少波动数据对测试结果的影响, 使结果更加稳定可靠。

5.4.2 实验验证

我们将方法 6 和方法 7 的各项特征总结在表 2 中。表 1 为 Matsui 方法(即 Fog 方法)、SUPERCOP 方法以及 Gladman 方法的各项特征。通过表 1 和表 2, 我们可以对以上 6 种方法的异同点做直观比较。

为进一步观察方法 6 和方法 7 的可靠性和稳定性, 我们分别将方法 7 的测试结果与 Matsui 方法(即 Fog 方法)的测试结果、SUPERCOP 方法的测试结果以及 Gladman 方法的测试结果进行比较。在方法 7 中, 采用方法 6 计算最小速度。

我们进行以下两组实验。

实验 8

分别采用 Matsui 方法(即 Fog 方法)、SUPERCOP 方法和方法 7, 测试 AES-128 加解密短消息的软件实现速度。

1) 采用 **Matsui 方法**(即 Fog 方法), 测试加解密函数的最小速度: foo 为 AES-128 加密函数(或解密函数); $x=1$; $loops=1000$; len 的取值为 128～4096B;

2) 采用 **SUPERCOP 方法**, 测试加解密函数的平均速度和标准差: foo 为 AES-128 加密函数(或解密函数); $x=1$; $loops=1000$; len 的取值为 128～4096B;

3) 采用**方法 7**, 测试加解密函数的最小速度、平均速度和标准差: foo 为 AES-128 加密函数(或解密函数); len 的取值为 128～4096B。

我们将实验结果整理在表 10 中。

表 10 实验 8 测试结果

Table 10 Results of experiment 8

	Enc				Dec			
	Matsui 方法 (Fog 方法)	SUPERCOP 方法	方法 7		Matsui 方法 (Fog 方法)	SUPERCOP 方法	方法 7	
len (Bytes)	*Ave* (cpb)	*Med* (cpb)	*Min* (cpb)	*Ave* (cpb)	*Ave* (cpb)	*Min* (cpb)	*Min* (cpb)	*Ave* (cpb)
128	13.96	15.63	13.48	13.92 (1.38%)	14.20	15.81	13.58	13.89 (1.70%)
256	12.88	13.41	12.29	12.49 (0.83%)	12.76	13.59	12.34	12.55 (1.50%)
384	12.68	12.69	11.91	12.08 (1.40%)	12.37	12.85	11.94	12.06 (0.88%)
512	12.32	12.37	11.80	11.94 (1.06%)	12.14	12.52	11.85	11.96 (1.05%)
640	12.09	12.13	11.65	11.78 (1.54%)	12.01	12.30	11.69	11.81 (1.29%)
768	11.80	11.98	11.53	11.64 (1.05%)	11.98	12.14	11.59	11.75 (1.55%)
896	12.12	11.87	11.48	11.67 (7.92%)	12.04	12.04	11.52	11.60 (1.28%)
1024	11.64	11.79	11.42	11.55 (1.42%)	12.98	11.96	11.48	11.55 (0.84%)
2048	11.67	11.50	11.25	11.43 (1.61%)	11.96	11.71	11.31	11.40 (1.01%)
3072	11.60	11.46	11.20	11.29 (1.01%)	11.69	11.61	11.26	11.34 (0.95%)
4096	11.69	11.38	11.18	11.27 (0.97%)	11.65	11.55	11.25	11.38 (1.17%)

(注: “*Med*”列中的数据表示所有 $cy[i]$ ($0 \leqslant i \leqslant loops-1$)的中位数 med, “*Min*”列中的数据表示最小速度 $v_{\min}$, “*Ave*”列中的数据表示平均速度 av, “*Ave*”列中括号中的百分数表示统计测试的标准差 sig。)

实验 9

分别采用 Gladman 方法和方法 7, 测试 AES-128 加解密短消息的软件实现速度。

1) 采用**方法 4**, 测试加解密函数的最小速度: foo 为 AES-128 加密函数(或解密函数); $x=8$; $loops=100$; len 的取值为 128～4096B;

2) 采用**方法 5**, 测试加解密函数的平均速度和标准差: foo 为 AES-128 加密函数(或解密函数); $y=16$; $loops_1=1000$, $loops_2=10000$; len 的取值为 128～4096B;

3) 采用**方法 7**, 测试加解密函数的最小速度、平均速度和标准差: foo 为 AES-128 加密函数(或解密函数); len 的取值为 128～4096B。

我们将实验结果整理在表 11 中。

观察表 10 和表 11, 分别将 Matsui 方法(即 Fog 方法)、SUPERCOP 方法和 Gladman 方法的测试结果与方法 7 的测试结果进行比较, 可以发现:

1) 利用方法 6 和方法 7 得到的测试结果更加稳定; 随着测试长度的变化, 测试结果的波动性非常小; 测试结果符合数据长度变长, 加解密速度变小的普遍规律;

2) 利用方法 7 得到的最小值和平均值之间具有

表 11 实验 9 测试结果

Table 11 Results of experiment 9

len (Bytes)	Enc				Dec			
	Gladman 方法结果		方法 7 结果		Gladman 方法结果		方法 7 结果	
	Min (cpb)	*Ave* (cpb)	*Min* (cpb)	*Ave* (cpb)	*Min* (cpb)	*Ave* (cpb)	*Min* (cpb)	*Ave* (cpb)
128	13.71	13.66(1.25%)	13.48	13.92 (1.38%)	13.95	13.86(1.02%)	13.58	13.89 (1.70%)
256	12.47	12.32(0.71%)	12.29	12.49 (0.83%)	12.64	12.53(0.84%)	12.34	12.55 (1.50%)
384	12.05	11.90(0.70%)	11.91	12.08 (1.40%)	12.25	12.11(0.74%)	11.94	12.06 (0.88%)
512	11.94	11.78(0.86%)	11.80	11.94 (1.06%)	12.11	11.95(0.72%)	11.85	11.96 (1.05%)
640	11.80	11.66(0.76%)	11.65	11.78 (1.54%)	11.97	11.83(0.60%)	11.69	11.81 (1.29%)
768	11.72	11.55(0.75%)	11.53	11.64 (1.05%)	11.89	11.75(0.77%)	11.59	11.75 (1.55%)
896	11.66	11.48(0.77%)	11.48	11.67 (7.92%)	11.83	11.67(0.73%)	11.52	11.60 (1.28%)
1024	11.61	11.43(0.69%)	11.42	11.55 (1.42%)	11.77	11.61(0.61%)	11.48	11.55 (0.84%)
2048	11.44	11.27(0.79%)	11.25	11.43 (1.61%)	11.63	11.45(0.69%)	11.31	11.40 (1.01%)
3072	11.41	11.22(0.80%)	11.20	11.29 (1.01%)	11.55	11.42(0.93%)	11.26	11.34 (0.95%)
4096	11.38	11.22(0.91%)	11.18	11.27 (0.97%)	11.52	11.41(0.89%)	11.25	11.38 (1.17%)

(注: “*Min*”列中的数据表示最小速度 $v_{\min}$, “*Ave*”列中的数据表示平均速度 *av*, “*Ave*”列中括号中的百分数表示统计测试的标准差 *sig* 。)

明显的关联性和可比性, 不会出现最小值比平均值大的情况;

3) 在方法 7 中, 我们将循环执行次数 $loops_2$ 的值适当减小为 1000, 以提高测试效率; 观察表 10 和表 11 可以发现, $loops_2$ 的值减小并没有影响测试数据的稳定性和可靠性。

综合上文的可行性分析和实验验证, 我们得出结论: 对于分组密码算法, **利用方法 6 和方法 7 得到的测试结果是可靠的, 稳定的, 高效的。**

在本节的最后, 我们需要说明的一点是: 利用方法 7 计算得到的平均值与最小值的数值是比较接近的。这是因为: 在正式采样 SAMPLE2 的取样过程中, 只有当标准差小于 30% 时, 平均值的测试数据才会被接受, 这样做可以排除测速过程中产生的不稳定数据。当标准差较小时, 数据达到稳定状态, 最小值和平均值是比较接近的(甚至是相等的)。

6 测试方法应用举例

本节应用第 5 节给出的方法 6 和方法 7, 测试 AES 算法和 SM4 算法加解密消息的软件实现速度。

6.1 AES 算法

6.1.1 AES-128

1) 测试 AES-128 加解密短消息的最小速度、平均速度和标准差。实验结果记录在表 12 中。

2) 测试 AES-128 加解密长消息的最小速度。实验结果记录在表 13 中。

表 12 AES-128 加解密短消息速度测试结果

Table 12 Testing results of encrypting and decrypting short messages with AES-128

len (Bytes)	Enc		Dec	
	Min (cpb)	*Ave* (cpb)	*Min* (cpb)	*Ave* (cpb)
128	13.38	13.69(0.87%)	13.6	13.94(1.01%)
256	12.20	12.36(0.90%)	12.40	12.56(0.83%)
512	11.70	11.80(0.81%)	11.84	11.99(1.34%)
1024	11.36	11.44(0.63%)	11.53	11.61(0.79%)
2048	11.20	11.25(0.60%)	11.37	11.43(0.63%)
3072	11.15	11.24(0.71%)	11.32	11.40(0.70%)
4096	11.13	11.19(0.63%)	11.31	11.35(0.48%)
5120	11.13	11.19(0.81%)	11.30	11.38(0.93%)
6144	11.13	11.20(1.07%)	11.30	11.38(0.94%)
8192	11.13	11.19(0.88%)	11.31	11.39(0.95%)

(注: “*Min*”列中的数据表示最小速度 $v_{\min}$, “*Ave*”列中的数据表示平均速度 *av*, “*Ave*”列中括号中的百分数表示统计测试的标准差 *sig* 。)

表 13 AES-128 加解密长消息速度测试结果

Table 13 Testing results of encrypting and decrypting long messages with AES-128

len (Mbytes)	Enc $v_{\min}$ (cpb)	Dec $v_{\min}$ (cpb)
1	11.33	11.46
2	11.34	11.50
3	11.32	11.47
4	11.32	11.47
5	11.35	11.50
6	11.35	11.51

6.1.2 AES-192

1) 测试 AES-192 加解密短消息的最小速度、平均速度和标准差。实验结果记录在表 14 中。

表 14 AES-192 加解密短消息速度测试结果

Table 14 Testing results of encrypting and decrypting short messages with AES-192

len (Bytes)	Enc		Dec	
	Min (cpb)	*Ave* (cpb)	*Min* (cpb)	*Ave* (cpb)
128	15.90	16.19(0.45%)	16.08	16.39(0.40%)
256	14.43	14.63(0.37%)	14.62	14.80(0.39%)
512	13.77	13.88(0.81%)	13.96	14.05(0.26%)
1024	13.39	13.46(0.23%)	13.54	13.60(0.19%)
2048	13.20	13.24(0.23%)	13.35	13.39(0.25%)
3072	13.13	13.18(0.30%)	13.28	13.32(0.27%)
4096	13.10	13.15(0.34%)	13.26	13.29(0.45%)
5120	13.09	13.14(0.45%)	13.25	13.28(0.50%)
6144	13.09	13.14(0.66%)	13.24	13.29(0.71%)
8192	13.09	13.15(0.86%)	13.25	13.29(0.87%)

(注: “*Min*”列中的数据表示最小速度 $v_{\min}$, “*Ave*”列中的数据表示平均速度 *av*, “*Ave*”列中括号中的百分数表示统计测试的标准差 *sig*。)

2) 测试 AES-192 加解密长消息的最小速度。实验结果记录在表 15 中。

表 15 AES-192 加解密长消息速度测试结果

Table 15 Testing results of encrypting and decrypting long messages with AES-192

len (Mbytes)	Enc $v_{\min}$ (cpb)	Dec $v_{\min}$ (cpb)
1	13.20	13.44
2	13.21	13.48
3	13.21	13.47
4	13.21	13.48
5	13.21	13.48
6	13.21	13.48

6.1.3 AES-256

1) 测试 AES-256 加解密短消息的最小速度、平均速度和标准差。实验结果记录在表 16 中。

2) 测试 AES-256 加解密长消息的最小速度。实验结果记录在表 17 中。

实验结果分析

由表 12 和表 13 的结果、表 14 和表 15 的结果、表 16 和表 17 的结果可以看出, 当数据长度超过 1024B 时, AES 算法加解密的速度达到稳定状态。

6.2 SM4 算法

1) 测试 SM4 加解密短消息的最小速度、平均速度和标准差。实验结果记录在表 18 中。

表 16 AES-256 加解密短消息速度测试结果

Table 16 Testing results of encrypting and decrypting short messages with AES-256

len (Bytes)	Enc		Dec	
	Min (cpb)	*Ave* (cpb)	*Min* (cpb)	*Ave* (cpb)
128	19.15	19.50(0.50%)	18.90	19.22(0.45%)
256	17.31	17.55(0.35%)	17.15	17.35(0.34%)
512	16.30	16.45(0.40%)	16.19	16.32(0.40%)
1024	15.68	15.78(0.28%)	15.63	15.74(0.51%)
2048	15.37	15.44(0.63%)	15.37	15.45(0.70%)
3072	15.26	15.36(0.96%)	15.27	15.37(0.89%)
4096	15.20	15.30(0.86%)	15.24	15.34(0.87%)
5120	15.19	15.29(0.89%)	15.23	15.31(0.78%)
6144	15.19	15.27(0.75%)	15.22	15.31(0.90%)
8192	15.16	15.31(1.41%)	15.21	15.33(1.42%)

(注: “*Min*”列中的数据表示最小速度 $v_{\min}$, “*Ave*”列中的数据表示平均速度 *av*, “*Ave*”列中括号中的百分数表示统计测试的标准差 *sig*。)

表 17 AES-256 加解密长消息速度测试结果

Table 17 Testing results of encrypting and decrypting long messages with AES-256

len (Mbytes)	Enc $v_{\min}$ (cpb)	Dec $v_{\min}$ (cpb)
1	15.34	15.43
2	15.34	15.46
3	15.34	15.45
4	15.34	15.46
5	15.34	15.47
6	15.33	15.47

表 18 SM4 加解密短消息速度测试结果

Table 18 Testing results of encrypting and decrypting short messages with SM4

len (Bytes)	Enc		Dec	
	Min (cpb)	*Ave* (cpb)	*Min* (cpb)	*Ave* (cpb)
128	28.40	28.62(1.17%)	28.70	28.91(1.05%)
256	26.67	26.82(0.74%)	26.77	27.02(0.82%)
512	25.78	25.91(0.74%)	25.81	26.14(0.72%)
1024	25.36	25.50(0.73%)	25.18	25.47(0.80%)
2048	25.14	25.29(0.78%)	25.14	25.44(0.82%)
3072	25.07	25.26(0.85%)	25.12	25.43(0.86%)
4096	25.06	25.25(0.81%)	25.13	25.42(0.88%)
5120	25.07	25.24(0.86%)	25.20	25.44(0.94%)
6144	25.08	25.25(0.84%)	25.21	25.44(0.86%)
8192	25.09	25.34(1.83%)	25.25	25.58(2.17%)

(注: “*Min*”列中的数据表示最小速度 $v_{\min}$, “*Ave*”列中的数据表示平均速度 *av*, “*Ave*”列中括号中的百分数表示统计测试的标准差 *sig*。)

2) 测试 SM4 加解密长消息的最小速度。实验结果记录在表 19 中。

表 19 SM4 加解密长消息速度测试结果

Table 19 Testing results of encrypting and decrypting long messages with SM4

len (Mbytes)	Enc $v_{\min}$ (cpb)	Dec $v_{\min}$ (cpb)
1	25.23	25.50
2	25.28	25.57
3	25.16	25.47
4	25.21	25.49
5	25.22	25.47
6	25.18	25.49

实验结果分析

由表 18 和 19 的结果可以看出, 当数据长度超过 1024B 时, SM4 算法加解密的速度达到稳定状态。

7 总结

在密码算法的设计和评估工作中, 测试密码算法的软件实现速度是一项必不可少的工作。在目前已有的工作中, 关于如何在 x64 平台上进行密码算法软件实现速度测试这一问题, 没有统一的测试规范。

本文首先对现有的在 x64 平台上对密码算法进行软件实现速度测试的四种方法(Matsui 方法、Fog 方法、SUPERCOP 方法、Gladman 方法)进行了介绍, 对四种方法的异同点进行了比较, 分析了四种方法中存在的问题。在此基础上, 本文对速度测试的公式选择、样本量大小的选择、影响速度测试结果稳定性的因素等内容都进行了理论分析和实验探究。本文以分组密码算法的速度测试为例, 通过理论分析和实验验证, 提出了在 x64 平台上测试分组密码算法软件实现最小速度的方法-方法 6 以及同时测试分组密码算法软件实现速度最小值、平均值和标准差的方法-方法 7。

利用方法 6 或方法 7 得到的密码算法速度测试的结果具有以下三个特性: 1) 测试结果是可靠的, 测试过程的取样量充分, 测试函数给出的测试速度是可信的; 2) 测试结果是稳定的, 测试函数给出的测试结果随机性小, 结果不会偏大或偏小; 3) 测试结果是高效的, 在保证测试结果可靠和稳定的前提下, 测试函数的取样量较小, 测试过程耗时较少。

方法 6 和方法 7 的应用范围不只局限于分组密码算法。在方法 6 和方法 7 中, 我们将待测试的密码算法设计为独立的输入模块, 测试函数的执行不依赖于具体的密码算法, 具有良好的普适性。利用本文给出的方法, 可以测试密码算法和方案 (包括但不局限于分组密码、流密码、公钥密码、杂凑函数、认证加密方案等) 在 x64 平台上的软件实现速度。

参考文献

[1] Announcing Request for Candidate Algorithm Nominations for the Advanced Encryption Standard. Federal Register: a Notice by the National Institute of Standards and Technology. http://www. federalregister.gov/documents/1997/09/12/97-24214/announcing-request-for-candidate-algorithm-nominations-for-the-advanced-encryption-standard. Sep. 1997.

[2] Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family. Federal Register: a Notice by the National Institute of Standards and Technology. http://www.federalregister.gov/documents/2007/11/02/E7-21581/announcing-request-for-candidate-algorithm-nominations-for-a-new-cryptographic-hash-algorithm-sha-3. Nov. 2007.

[3] Lightweight Cryptography (LWC) Standardization Process. National Institute of Standards and Technology. http://csrc.nist.gov/Projects/Lightweight-Cryptography. Aug. 2020.

[4] Notice of the National Cryptographic Algorithm Design Competition. Chinese Association for Cryptologic Research. http://www.cacrnet.org.cn/site/content/259.html. Jun. 2018.
(关于举办全国密码算法设计竞赛的通知. 中国密码学会. http://www.cacrnet.org.cn/site/content/259.html. 2018 年 6 月.)

[5] Submission Requirements and Evaluation Criteria for the Lightweight Cryptography Standardization Process. National Institute of Standards and Technology. http://csrc.nist.gov/CSRC/media/on-requirements-august2018.pdf. Aug. 2018.

[6] Notice of the National Cryptographic Algorithm Design Competition, Attached File 1: Technical Requirements for the Cryptographic Algorithm Design Competition. Chinese Association for Cryptologic Research. http://www.cacrnet.org.cn/site/content/259. html. Jun. 2018.
(全国密码算法设计竞赛通知, 附件 1: 密码算法设计竞赛技术要求. 中国密码学会. http://www.cacrnet.org.cn/site/content/259.html. 2018 年 6 月.)

[7] FELICS – Fair Evaluation of Lightweight Cryptographic Systems. Lightweight Cryptography Workshop 2015 held by the National Institute of Standards and Technology. http://csrc.nist.gov/csrc/ media/events/lightweight-cryptography-workshop-2015/documents/papers/session7-dinu-paper.pdf. Sep. 2016.

[8] Matsui M, Fukuda S. How to Maximize Software Performance of Symmetric Primitives on Pentium III and 4 Processors[C]. *The 12th international conference on Fast Software Encryption*, 2005: 398-412.

[9] Software Optimization Resources: Test Programs for Measuring Clock Cycles and Performance Monitoring. The Research Website of A. Fog. http://www.agner.org/optimize/. Aug. 2018.

[10] eBACS: ECRYPT Benchmarking of Cryptographic Systems. D.J.Bernstein, T.Lange (editors). http://bench.cr.yp.to/. Feb. 2020.

[11] The Website of SUPERCOP. eBACS: ECRYPT Benchmarking of Cryptographic Systems. http://bench.cr.yp.to/supercop.html. Aug. 2020.

[12] B.Gladman's Program of Testing the Performance of AES. The Github Website. http://github.com/BrianGladman/AES. Dec. 2019.

[13] Keccak: Open-source Cryptography. Team Keccak: G. Berton, J. Daemen, M. Peeters, G.V. Assche. http://keccak.noekeon.org/. Sep. 2017.

[14] The Website of ASCON: the Lightweight Authenticated Encryption & Hashing. C. Dobraunig, M. Eichlseder, F. Mendel, et al (authors). http://ascon.iaik.tugraz.at/. 2019.

[15] Advanced Encryption Standard Specified in FIPS 197. National Institute of Standards and Technology. http://csrc.nist.gov/glossary/term/AES. Nov. 2001.

[16] Banik S, Pandey S K, Peyrin T, et al. GIFT: A Small Present[M]. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017: 321-345.

[17] A Brief Introduction of Intrinsic Functions: __rdtsc(). Microsoft Docs. http://docs.microsoft.com/en-us/cpp/intrinsics/rdtsc?view=vs-2019. Nov. 2016.

[18] Knudsen L, Robshaw M. The block cipher companion[M]. Heidelberg: Springer-Verlag Berlin Heidelberg, 2011.

[19] Daemen J, Rijmen V. The Design of Rijndael: The Advanced Encryption Standard (AES)[M]. Berlin, Heidelberg: Springer Berlin Heidelberg, 2020.

[20] Benadjila R, Guo J, Lomné V, et al. Implementing Lightweight Block Ciphers on X86 Architectures[C]. *Revised Selected Papers on Selected Areas in Cryptography—SAC 2013 - Volume 8282*, 2013: 324-351.

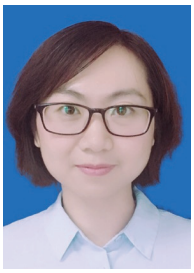
[21] The SM4 Cryptographic Algorithm Using in the Wireless LAN Products. State Cryptography Administration. http://www.sca.gov.cn/sca/c100061/201611/1002423/files/330480f731f64e1ea75138211ea0dc27.pdf. Nov. 2016.
(无线局域网产品使用的 SM4 密码算法. http://www.sca.gov.cn/sca/c100061/201611/1002423/files/330480f731f64e1ea75138211ea0dc27.pdf. 国家密码管理局. 2016 年 11 月.)

[22] Implementations of Low Cost Block Ciphers in Atmel AVR Devices. ECRYPT II website. http://perso.uclouvain.be/fstandae/source_codes/lightweight_ciphers/. 2012.

[23] Implementations of Hash Functions in Atmel AVR Devices. ECRYPT II website. http://perso.uclouvain.be/fstandae/source_codes/hash_atmel/. 2012.

季福磊 于 2016 年在山东大学信息安全专业获得学士学位。现在中国科学院信息工程研究所网络空间安全专业硕博连读。研究领域为对称密码算法安全性分析。Email: jifulei@iie.ac.cn

张文涛 于 2004 年在中国科学院软件研究所获得博士学位。现为中国科学院信息工程研究所国家重点实验室研究员。研究领域为对称密码算法的设计与分析。Email: zhangwentao@iie.ac.cn



毛颖颖 于 2009 年在西安电子科技大学大学密码学专业获得硕士学位。现任国家密码管理局商用密码检测中心检测工程师。研究领域为密码算法测评、密码产品测评。Email: maoyy2000@163.com

赵雪锋 于 2017 年在山西大学计算机科学与技术专业获得学士学位。现在中国科学院信息工程研究所网络空间安全专业硕博连读。研究领域为对称密码算法安全性分析。Email: zhaoxuefeng@iie.ac.cn