DOI:10.11992/tis.201507036
网络出版地址:http://www.cnki.net/kcms/detail/23.1538.tp.20151110.1354.004.html

SM3 杂凑算法的软件快速实现研究

杨先伟[1],康红娟[2]
(1. 无锡职业技术学院 基础部,江苏 无锡 214121; 2. 四川长虹电器股份有限公司,四川 成都 610041)

摘 要:杂凑算法是密码学中最基本的模块之一,可广泛应用于密码协议、数字签名、消息鉴别等领域。我国国家密码管理局在 2010 年发布了 SM3 密码杂凑算法,该算法适用于商用密码应用中的数字签名和验证、消息认证码的生成与验证以及随机数的生成等。该文重点研究 SM3 密码杂凑算法的软件快速实现,根据算法本身的特点,尤其是压缩函数的特点,给出一种更加适用于软件的快速实现方式。实验表明利用此方法可以将算法的效率提升 60%左右。

关键词:SM3 算法;杂凑函数;软件快速实现; 数字签名;消息鉴别;完整性认证;数字指纹; 压缩函数
中图分类号:TP309 **文献标志码**:A **文章编号**:1673-4785(2015)06-0954-06

中文引用格式:杨先伟,康红娟. SM3 杂凑算法的软件快速实现研究[J]. 智能系统学报, 2015, 10(6): 954-959.
英文引用格式:YANG Xianwei, KANG Hongjuan. Fast software implementation of SM3 Hash algorithm[J]. CAAI Transactions on Intelligent Systems, 2015, 10(2): 954-959.

Fast software implementation of SM3 Hash algorithm

YANG Xianwei[1], KANG Hongjuan[2]
(1. Department of Fundamental Courses, Wuxi Prof Technology inst., Wuxi 214121, China; 2. Sichuan Changhong Electric Co., Ltd., Chengdu 610041, China)

Abstract: The hash algorithm is one of the most basic cryptography modules, and is widely used in cryptographic protocols, digital signatures, message authentication, and in other fields. The Chinese National Cryptography Administration released the SM3 hash algorithm in 2010. This algorithm is applied to digital signature and verification, the generation and verification of message authentication codes, and random number generation. This paper addresses the fast software implementation of the SM3 algorithm. Based on the SM3 features, and especially its compression function characteristics, we propose a method that is highly suitable for fast software implementation. Experimental results show that this method can improve the implementation speed by 60%.

Keywords: SM3 algorithm; hash function; fast software implementation; digital signature; message authentication; integrity authentication; digital fingerprint; compression function

哈希(Hash)函数,也叫杂凑函数,是密码学中最基本的模块之一,广泛应用于密码协议、数字签名、消息鉴别、完整性认证等领域。因此,它在密码学中扮演着极其重要的角色。

杂凑函数的目的是产生数据块的“指纹”,它可以对任意长度的信息产生定长的输出。这个变换过程软硬件均易于计算实现,但其逆向变换过程在计算上不可行,即具有单向性。出于安全性的考虑,杂凑函数还必须满足抗弱碰撞性和抗强碰撞性。1991 年,Ron Rivest 提出了 MD5 算法,这曾经是使用最为广泛的杂凑算法。从 20 世纪 90 年代年开始,美国国家标准与技术研究院(NIST)陆续公布了 SHA 系列[1],并通过公开竞赛方式征集 SHA-3[2]。

中国国家密码管理局在 2010 年发布了 SM3 密

收稿日期:2015-07-23. 网络出版日期:2015-11-10.
基金项目:国家自然科学基金资助项目(11471144).
通信作者:杨先伟. E-mail:yangxianwei2018@163.com.

码杂凑算法[3],该算法适用于商用密码中的多种应用,满足多种密码应用的安全需求:1)数字签名和验证,如作为 SM2 算法中数字签名所需的杂凑函数;2)消息认证码的生成与验证,消息认证码不仅可以使用分组密码算法基于特定的工作模式生成,也可以使用 SM3 等杂凑函数生成;3)随机数的生成。

哈希函数在各种平台和环境下的执行效率是非常重要的考量指标之一,比如服务器端常需执行的 SSL/TLS 协议就使用了哈希函数进行认证。目前已有大量文章对 SHA 系列算法的软件快速实现进行研究,比如 Aciiçmez[4] 提出基于 SIMD 技术快速实现哈希算法,Gueron 等[5-6] 对并行处理多个消息的情况进行了研究。同样也有大量文章对 SM3 算法的软硬件快速实现进行研究。张倩等[7] 提出了一种 ASIC 高效实现架构;王晓燕等[8] 基于 FPGA 设计 SM3 算法 IP 核的整体架构,对关键逻辑进行优化设计;伍娟[9] 以同方公司 THD86 智能卡芯片为硬件平台实现了 SM3 算法;曾小波等[10] 分析了基于 8051 软核的 SM3 算法 IP 原理、设计流程及实现方案,该方案在时序和面积上均做到相当程度的优化,并提高了算法的效率;沈一公等[11] 基于 Android 平台研究了 SM3 算法的快速实现,并以此为基础研究文件防篡改以便检查手机软件的安装;易叔贤等[12] 结合已经将 SM 系列算法纳入其中的 PBOC 3.0 新规范,分析考虑 SM2、SM3、SM4 算法在金融 IC 卡领域的实现和应用。

与这些研究相比,文本研究的侧重点是 SM3 算法在普通软件平台下的快速实现方式。文本根据算法以及压缩函数的特点,给出一种更加适用于软件快速实现的算法描述方式和实现方法,本文提出的实现方法具有以下优点:首先,此方法避免了普通实现中可能采用的效率较低的实现架构和运算方式,可较大地提高算法的软件效率,经多个软件平台对比测试,本文的实现方法可将算法效率提升 60%左右;其次,此方式不基于特定的软件平台、架构、指令等,具有很强的跨平台性和兼容性。

1　SM3 算法简介

SM3 杂凑算法可将长度小于 2^{64} 比特的消息经过填充、反复的消息扩展和压缩,生成长度为 256 比特的杂凑值。在 SM3 算法中,字表示长度为 32 的比特串。

1.1　函数

布尔函数 $FF_i(X,Y,Z)$、$GG_i(X,Y,Z)$,$0 \leqslant i \leqslant 63$ 的定义如下:

$$FF_i(X,Y,Z) = \begin{cases} X \oplus Y \oplus Z, 0 \leqslant i \leqslant 15 \\ (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z), 16 \leqslant i \leqslant 63 \end{cases}$$

$$GG_i(X,Y,Z) = \begin{cases} X \oplus Y \oplus Z, 0 \leqslant i \leqslant 15 \\ (X \wedge Y) \vee (\neg X \wedge Z), 16 \leqslant i \leqslant 63 \end{cases}$$

置换函数 $P_0(X)$ 和 $P_1(X)$ 的定义如下:

$$P_0(X) = X \oplus (X <<< 9) \oplus (X <<< 17)$$

$$P_1(X) = X \oplus (X <<< 15) \oplus (X <<< 23)$$

1.2　填充

设消息的长度为 l 比特。填充方式为:首先将比特"1"添加到消息的末尾;然后添加 k 个"0",k 是满足 $l + 1 + k = 448 \bmod 512$ 的最小的非负整数;最后再将消息长度 l 的 64 位二进制表示添加在最末。填充后的消息比特长度为 512 的倍数。

1.3　迭代压缩

填充后的消息 m' 按 512 比特进行分组:$m' = B^{(0)} \| \cdots \| B^{(n-1)}$.对每个分组利用压缩函数 CF 进行迭代:

FOR i = 0 TO n − 1

$V^{(i+1)} \leftarrow CF(V^{(i)}, B^{(i)})$

END FOR

1.4　压缩函数

压缩函数 CF 的计算过程如下:

首先,计算消息扩展字 W_i,$0 \leqslant i \leqslant 67$ 和 W_i',$0 \leqslant i \leqslant 63$,步骤如下:

$W_0 \| \cdots \| W_{15} = B^{(i)}$

FOR i = 16 TO 67

$W_i \leftarrow P_1(W_{i-16} \oplus W_{i-9} \oplus (W_{i-3} <<< 15)) \oplus (W_{i-13} <<< 7) \oplus W_{i-6}$

END FOR

FOR i = 0 TO 63

$W'_i \leftarrow W_i \oplus W_{i+4}$

END FOR

然后,进行包含 64 轮迭代的压缩,步骤如下:

$A \| B \| C \| D \| E \| F \| G \| H \leftarrow V^{(i)}$

FOR i = 0 TO 63

$SS_1 \leftarrow ((A <<< 12) + E + (T_j <<< j)) <<< 7$

$SS_2 \leftarrow SS_1 \oplus (A <<< 12)$

$TT_1 \leftarrow FF_i(A,B,C) + D + SS_2 + W_i'$

$TT_2 \leftarrow GG_i(E,F,G) + H + SS_1 + W_i$

$D \leftarrow C$

$C \leftarrow B <<< 9$

$B \leftarrow A$

$A \leftarrow TT_1$

$H \leftarrow G$

$G \leftarrow F <<< 19$

$F \leftarrow E$

$E \leftarrow P_0(TT_2)$

END FOR

$V^{(i+1)} \leftarrow V^{(i)} \oplus (A \| B \| C \| D \| E \| F \| G \| H)$

1.5 输出杂凑值

256 比特杂凑值 y 的计算方式为

$y \leftarrow (A \| B \| C \| D \| E \| F \| G \| H) \leftarrow V^{(n)}$

2 软件快速实现

从理论上讲,SM3 算法中使用最多且最耗时的是 64 轮压缩函数和消息扩展。利用 Intel VTune Amplifier XE 分析算法热点,得出信息如下表。

表 1 普通实现时的热点

Table 1 The hot spots of general implementation

参数	耗时/s	百分比/%
整体	21.795	100
压缩函数	14.355	65.9
消息扩展	5.287	24.3
其他	2.153	9.8

热点信息显示,压缩函数和消息扩展的确是最耗时的 2 个部分,其耗时分别占总耗时的 65.9% 和 24.3%。因此,快速实现的关键在怎样快速实现压缩函数和消息扩展。

2.1 消息扩展的快速实现

消息扩展的目的是利用 512 比特的消息分组 B 扩展得到 68 个字 $W_0,\cdots,W_{67}$ 和 64 个字 $W'_0,\cdots,W'_{63}$。

快速实现时,为了尽可能减少不必要的数据加载和存储, $W_0,\cdots,W_{67}$ 和 $W'_0,\cdots,W'_{63}$ 的计算可以调整到压缩函数里执行,具体实施过程是:

1)首先在执行 64 轮压缩函数前只计算初始的 4 个字 $W_0,\cdots,W_3$;

2)然后在压缩函数的第 i 轮生成 W_{i+4},而 W'_i 则使用 $W'_i = W_i \oplus W_{i+4}$ 代替。

经过这样的调整,去掉了字 $W'_0,\cdots,W'_{63}$,减少了字 $W_0,\cdots,W_{67}$ 和 $W'_0,\cdots,W'_{63}$ 的加载和存储次数,提高了消息扩展的速度。

2.2 压缩函数的快速实现

压缩函数的快速实现可以从结构调整、流程变更、常数计算等方面着手。

1)压缩函数的结构可以做适当的调整。压缩函数每一轮的最末会执行如下所示的循环右移, $A \| B \| C \| D \leftarrow (A \| B \| C \| D) >>> 32$, $E \| F \| G \| H \leftarrow (E \| F \| G \| H) >>> 32$。为了减少循环移位导致的不必要的赋值运算,可以将字的循环右移变更每轮输入字顺序的变动,且这个顺序变动会在 4 轮后还原,具体情况如下(以下用 OneRound(·) 表示一轮压缩):

OneRound($i+0,A,B,C,D,E,F,G,H,W$)

OneRound($i+1,D,A,B,C,H,E,F,G,W$)

OneRound($i+2,C,D,A,B,G,H,E,F,W$)

OneRound($i+3,B,C,D,A,F,G,H,E,W$)

2)可以优化压缩函数的中间变量的生成流程。此优化生成流程可以去除不必要的赋值,减少中间变量个数。优化后的执行步骤如下(其中 $t_i = T_i <<< i$ 为常数):

$$\mathrm{TT}_2 \leftarrow A <<< 12$$

$$\mathrm{TT}_1 \leftarrow TT_2 + E + t_i$$

$$\mathrm{TT}_1 \leftarrow TT_1 <<< 7$$

$$\mathrm{TT}_2 \leftarrow TT_2 \oplus TT_1$$

3)利用上述调整以及消息扩展部分的调整可以将原来计算 TT_1、TT_2、D 和 H 的过程进行如下的进一步简化。

$$D \leftarrow D + FF_i(A,B,C) + TT_2 + (W_i \oplus W_{i+4})$$

$$H \leftarrow H + GG_i(A,B,C) + TT_1 + W_i$$

4)预先计算并存储常数 $t_i = T_i <<< i$。这可以避免每个消息分组都去计算常数,且占用的存储空间也很少,仅 256 Byte。

2.3 调整后的算法描述

优化后的算法将消息扩展和压缩函数结合在一起。下面先描述调整后的消息处理算法,该算法完成消息扩展和 64 轮压缩迭代;再描述调整后的一轮算法,该算法完成一轮压缩迭代,包括计算必需的消息扩展字 W_{i+4}。调整后的消息处理算法描述如下。

算法 1 调整后的消息处理算法

ProcessBlock(V,M)

输入:上轮迭代结果 V,一个消息分组 B

输出:本轮迭代结果 V

中间变量:字寄存器 A—H,

步骤:

1) $W_0 \| W_1 \| W_2 \| W_3 \leftarrow B_0 \| B_1 \| B_2 \| B_3$,

2) $A \| B \| C \| D \| E \| F \| G \| H \leftarrow V$,

3) FOR ($i = 0,4,8,\cdots,60$),

OneRound($i+0,A,B,C,D,E,F,G,H,W$),

OneRound($i+1,D,A,B,C,H,E,F,G,W$),

OneRound($i+2,C,D,A,B,G,H,E,F,W$),

OneRound($i+3,B,C,D,A,F,G,H,E,W$),

END FOR

4) $V \leftarrow V \oplus (A \| B \| C \| D \| E \| F \| G \| H)$,

5)返回 V。

对算法 1 做以下几点说明:这里的 $B_0 \| B_1 \| \cdots \| B_{15} = B$ 分别代表消息的 16 个字;前 4 个消息扩展字 W_0、W_1、W_2、W_3 需在循环前计算出来,进入后面的循环后,每次执行 OneRound(i, $*$) 将计算 W_{i+4}。

调整后的一轮压缩算法如下。

算法 2　调整后的一轮压缩算法

OneRound(i,A,B,C,D,E,F,G,H,W)

输入:字寄存器 A—H, 轮序号 i,消息扩展字数组 $W = (W_0,\cdots,W_{67})$

输出:更新后的 A—H 和 $W = (W_0,\cdots,W_{67})$

步骤:

1)计算消息扩展字 W_{i+4}

IF($i < 12$) $W_{i+4} \leftarrow B_{i+4}$

ELSE　$W_{i+4} \leftarrow P_1(W_{i-12} \oplus W_{i-5} \oplus (W_{i+1} <<< 15)) \oplus (W_{i-9} <<< 7) \oplus W_{i-2}$

END IF

2)计算中间变量 TT_1 和 TT_2

$$TT_2 \leftarrow A <<< 12$$

$$TT_1 \leftarrow TT_2 + E + t_i$$

$$TT_1 \leftarrow TT_1 <<< 7$$

$$TT_2 \leftarrow TT_2 \oplus TT_1$$

3)仅更新字寄存器 B、D、F、H。

$$D \leftarrow D + FF_i(A,B,C) + TT_2 + (W_i \oplus W_{i+4})$$

$$H \leftarrow H + GG_i(E,F,G) + TT_1 + W_i$$

$$B \leftarrow B <<< 9$$

$$F \leftarrow F <<< 19$$

$$H \leftarrow P_0(H)$$

4)返回更新后的 A—H 和 $W = (W_0,\cdots,W_{67})$。

对算法 2 做以下几点说明:进入第 i 轮的算法 2 之时,消息扩展字只有 $\{W_k \mid k < i+4\}$ 这部分信息已经求出,执行完毕后 W_{i+4} 也被计算出来;步骤 2 中的 t_i 为常量 $T_i <<< i$,应预先计算并存储,使用时只需查表;由于 W_i、FF_i、GG_i 的计算方式在 $i < 16$ 时和 $i \geqslant 16$ 时不同,因此可以考虑将 OneRound 函数分为 $0 \leqslant i < 12$、$12 \leqslant i < 16$、$16 \leqslant i < 643$ 种情况分别实现。

3　2 种实现方式的计算量分析评估

为了从理论上评估新方法的效率,本节对 2 种方法的计算量进行详细对比。由于算法的操作主要集中在压缩函数中,因此以下对压缩函数的计算量进行统计、分析和对比。优化前的方法严格按照标准文档,先计算消息扩展字,再进行 64 轮迭代,优化后的方法则按照上一节描述的算法 1 和算法 2 进行实现。以下用 LOAD 和 STORE 表示数据加载和存储,XOR 表示异或运算,ROT 表示移位运算,ADD 表示加法运算,AND 表示与运算,OR 表示或运算,NOT 表示非运算。

优化前的算法中,消息扩展的计算量为:

1)计算前 16 个 W_i 时每个需执行 1 次 LOAD 和 1 次 STORE,计算后 52 个 W_i 时每个需执行 5 次 LOAD、1 次 STORE、6 次 XOR、4 次 ROT;

2)计算 64 个 W'_i 每个需执行 2 次 LOAD、1 次 STORE、1 次 ROT;

3)计算压缩函数的一次迭代需要执行 3 次 LOAD、12 次 STORE、8 次 ADD、3 次 XOR、8 次 ROT、1 次 FFi 函数和 1 次 GGi 函数,

4)FFi 函数和 GGi 函数的计算量是,前 16 次 FFi 函数需执行 2 次 XOR 和 2 次 ROT,前 16 次 GGi 函数需执行 2 次 XOR 和 2 次 ROT,后 48 次 FFi 函数需执行 3 次 AND 和 2 次 OR,后 48 次 GGi 函数需执行 2 次 AND、1 次 OR、1 次 NOT。

根据以上统计分析,表 2 列出了优化前的算法中对一个 512 比特的消息块执行一次完整的压缩所需的计算量。

优化后的算法中,消息扩展的计算量为:

1)计算前 12 个 W_{i+4} 时每个需执行 1 次 LOAD 和 1 次 STORE,计算后 52 个 W_{i+4} 时每个需执行 5 次 LOAD、1 次 STORE、6 次 XOR、4 次 ROT;

2)计算中间变量 TT_1 和 TT_2 需要执行 1 次 LOAD、2 次 STORE、2 次 ADD、1 次 XOR、2 次 ROT;

3)更新字寄存器 B、D、F、H 需要执行:1 次 LOAD、1 次 STORE、6 次 ADD、3 次 XOR、4 次 ROT、

1 次 FFi 函数和 1 次 GGi 函数;

4) FFi 函数和 GGi 函数的计算量同优化前的计算量。

根据以上统计分析,表 2 列出了优化后的算法中对一个 512 比特的消息块执行一次完整的压缩所需的计算量。

表 2 优化前后一次压缩函数的计算量

Table 2 The computation of the compression function of the before and after optimization

	LOAD	STORE	ADD	XOR	ROT	AND	OR	NOT	合计
优化前	596	900	512	632	720	240	144	48	3 792
优化后	400	256	512	632	592	240	144	48	2 824

从表 2 可知,优化后的压缩函数通过轮函数的调整和消息扩展函数的优化,大大减少了 LOAD 和 STORE 的次数,同时中间变量 TT_1 和 TT_2 的优化实现又进一步减少了 ROT 的次数,其余运算的计算量无变化。

如果从操作总数的角度考虑,优化后算法的速度可提升 (3 792 − 2 824)/2 824 = 34.3% 。但实际上 CPU 执行这些操作指令时,不同的操作具有不同的指令执行周期(cycle),甚至不同的 CPU 执行相同的运算所需的指令周期也各不相同。大部分 CPU 执行整数的算数运算和逻辑运算需 1 个时钟周期,而执行 LOAD 和 STORE 则需要多个时钟周期,且各 CPU 的执行时间也有较大差异。以下假设执行每个算数逻辑运算需 1 个时钟周期。如果执行 LOAD 需 1 个时钟周期,执行 STORE 需 2 个时钟周期,则优化后算法的速度可提升 52.3%。;如果假设执行 LOAD 需 1.5 个时钟周期,执行 STORE 需 2.5 个时钟周期,则优化后算法的速度可提升 59.6%;如果假设执行 LOAD 需 2 个时钟周期,执行 STORE 需 3 个时钟周期,则优化后算法的速度可提升 65.6%。不同假设下的速度提升情况见下表 3。

表 3 不同情况下的优化前后速度提升估计值

Table 3 The speed of the before and after optimization

算数逻辑运算(cycle)	LOAD (cycle)	STORE (cycle)	速度提升/%
1	1.0	1.0	34.3
1	1.0	2.0	52.3
1	1.5	2.5	59.6
1	2.0	3.0	65.6

4 模拟实验与对比测试

为了模拟真实环境中对 SM3 算法软件实现的需求,下面的实验中进行了 4 组测试,每组测试方法对多个数据包进行杂凑,每个数据包为特定长度字节,然后统计耗时和速度。以下为 4 组测试的详细情况说明。

1) 第 1 组测试中测试 1 个数据包,该数据包为 256×10^6 个字节,此测试用以模拟大量数据杂凑的情况,如大型文件杂凑;

2) 第 2 组测试中杂凑 200 个数据包,每个数据包 1.28×10^6 个字节,此测试用以模拟中型数据包杂凑的情况,如图片等;

3) 第 3 组测试中杂凑 40 000 个数据包,每个数据包 6.4×10^3 个字节,此测试用以模拟普通网络数据包杂凑的情况;

4) 第 4 组测试中杂凑 8×10^6 个数据包,每个数据包 32 个字节,此测试用以模拟频繁的微小型数据包杂凑的情况。

为了统计每种测试的准确耗时值,每组测试都反复进行 21 次并记录各次的时间,最后从大到小排列后取最中间的值作为统计耗时值。

测试使用的软件平台详情如下:Windows XP SP3 32 比特、Intel Core i3@ 3400 MHz、4 GB DDR3-1600 SDRAM、Microsoft Visual Studio 8.0。速度单位为 Mbit/s。其中处理器的缓存情况为[13]:一级缓存为每个核心 32 KB,2 级缓存为每个核心 64 KB,3 级缓存为多核共享 3 MB。

表 4 2 种实现方式的性能比较

Table 4 The performance comparison of two implementation methods

测试类别	速度/(Mb · s^{-1})		速度提升/%
	优化前	优化后	
第 1 组测试	739	1 203	62.8
第 2 组测试	733	1 191	62.5
第 3 组测试	701	1 074	53.2
第 4 组测试	642	973	51.6
平均	704	1 110	57.7

上表列出的测试结果表明:1) 数据包越大,执行效率越高,这是因为大型数据包减少了一头一尾的初始化、消息填充和反初始化等工作;2) 优化调整后

的算法效率提升显著,可以提升 60%左右,在杂凑大中型数据包时速度提升 60%以上,即使在杂凑微小型数据包时效率也能提升 50%以上。

4　结束语

本文对我国国家密码管理局发布的 SM3 密码杂凑算的软件快速实现进行研究,根据算法自身的特点,尤其是压缩函数的特点,给出一种更加适用于软件快速实现的算法描述方式和实现方法。理论分析得出的算法计算量以及模拟实验结果均表明,利用此软件快速实现方法可以将算法的效率提升 60%左右。另外,此软件快速实现方式不基于特定的平台、架构、指令等,因此具有很强的跨平台性和兼容性。

参考文献:

[1] NIST. Federal information processing standards publication 180-3, secure hash standards (SHS) [S]. Gaithersburg, MD, USA: Information Technology Laboratory of National Institute of Standards and Technology, 2008. http://csrc.nist.gov/publications.

[2] NIST. Cryptographic hash algorithm competition [EB/OL]. (2005-04-15)[2015-08-05]. http://csrc.nist.gov/groups/ST/hash/sha-3/index.html.

[3] 国家密码管理局. SM3 密码杂凑算法[S]. 北京: 国家密码管理局, 2010.
National Cryptography Administration. SM3 cryptographic hash algorithm[S]. Beijing: National Cryptography Administration, 2010.

[4] ACIICMEZ O. Fast hashing on pentium SIMD architecture [D]. Corvallis, Oregon: Oregon State University, 2004.

[5] GUERON S, KRASNOV V. Parallelizing message schedules to accelerate the computations of hash functions[R]. 2012. http://eprint.iacr.org/2012/067.pdf

[6] GUERON S, KRASNOV V. Simultaneous hashing of multiple messages[J]. Journal of Information Security, 2012, 3(4): 319-325.

[7] 张倩, 李树国. SM3 杂凑算法的 ASIC 设计和实现[J]. 微电子学与计算机, 2014, 31(9): 143-146, 152.
ZHANG Qian, LI Shuguo. Design and implementation of SM3 algorithm in ASIC[J]. Microelectronics & Computer, 2014, 31(9): 143-146, 152.

[8] 王晓燕, 杨先文. 基于 FPGA 的 SM3 算法优化设计与实现[J]. 计算机工程, 2012, 38(6): 244-246.
WANG Xiaoyan, YANG Xianwen. Optimization design and implementation of SM3 algorithm based on FPGA[J]. Computer Engineering, 2012, 38(6): 244-246.

[9] 伍娟. 国密 SM3 算法在 COS 上的研究与实现[J]. 科技信息, 2013, (2): 294-295.
WU Juan. Research and implementation of SM3 algorithm on COS[J]. Science & Technology Information, 2013, (2): 294-295.

[10] 曾小波, 唐忠彪, 焦歆. 基于单片机的 SM3 算法优化及 Verilog 模型验证[J]. 电子科技, 2015, 28(2): 38-40.
ZENG Xiaobo, TANG Zhongbiao, JIAO Xin. Optimization of SM3 algorithm and Verilog model validation based on SCM[J]. Electronic Science and Technology, 2015, 28(2): 38-40.

[11] 沈一公, 苏厚勤. 基于 Android 的 SM3 密码杂凑算法研究与实现[J]. 电子技术与软件工程, 2013(18): 69-70.
SHEN Yigong, SU Houqin. Research and implementation of SM3 algorithm based on android[J]. Electronic Technology & Software Engineering, 2013(18): 69-70.

[12] 易叔贤, 张非凡. SM 系列算法在金融 IC 卡领域的应用[J]. 金融电子化, 2013(7): 49-52.
YI Shuxian, ZHANG Feifan. Application of SM series algorithm in the field of financial IC card[J]. Financial Computerizing, 2013(7): 49-52.

[13] ntel. 2nd generation intel® coreTM processor family desktop datasheet[EB/OL]. (2011-01-04)[2013-07-08]. http://www.intel.com/content/www/us/en/processors/core/2nd-gen-core-desktop-vol-1-datasheet.html.

作者简介:

杨先伟,男,1980 年生,讲师,主要研究方向为通信与系统工程。

康红娟,女,1983 年生,工程师,主要研究方向为保密通信。