暗号アルゴリズムの最新動向
ー安全性と実装性の現状と課題ー

2006年12月7日

三菱電機情報技術総合研究所
松井　充

暗号技術でできること

- **秘匿** (Confidentiality)
  - **暗号化**
    - **通信路の盗聴を防止**
- **完全性** (Integrity)
  - **認証子の付加／検証**
    - **メッセージの改ざん防止**
- **ユーザ認証** (Authentication)
  - Challenge and Response
    - **なりすましの防止**

暗号方式の分類

共通鍵暗号（秘密鍵暗号、対称鍵暗号）

ストリーム暗号	RC4
ブロック暗号	(Triple)DES, AES, MISTY/KASUMI

公開鍵暗号（非対称鍵暗号）

素因数分解型暗号	RSA
離散対数型暗号	ElGamal, Diffie-Hellman
楕円離散対数型暗号	EC-ElGamal, EC-Diffie-Hellman

暗号の安全性とは何か（1）

- 安全な暗号 = 解読が困難な暗号
  - 解読とは何か、困難とは何かは「定義次第」
  - それらは先験的に存在する概念ではない
- 解読とは何か
  - 例えば：　平文を求めること
  - 例えば：　暗号の鍵を求めること
  - 例えば：　署名を偽造すること

解読 ＝ “attacker” が主人公の game の goal

暗号の安全性とは何か (2)

- 解読困難／解読可能とは何か
  - 共通鍵暗号の場合
    - 鍵の全数探索よりも高速な鍵導出アルゴリズムの存在を意味することが多い (すべてではない)
  - 公開鍵暗号の場合
    - 法パラメータの大きさに対して多項式時間での鍵あるいは平文導出アルゴリズムの存在を意味することが多い

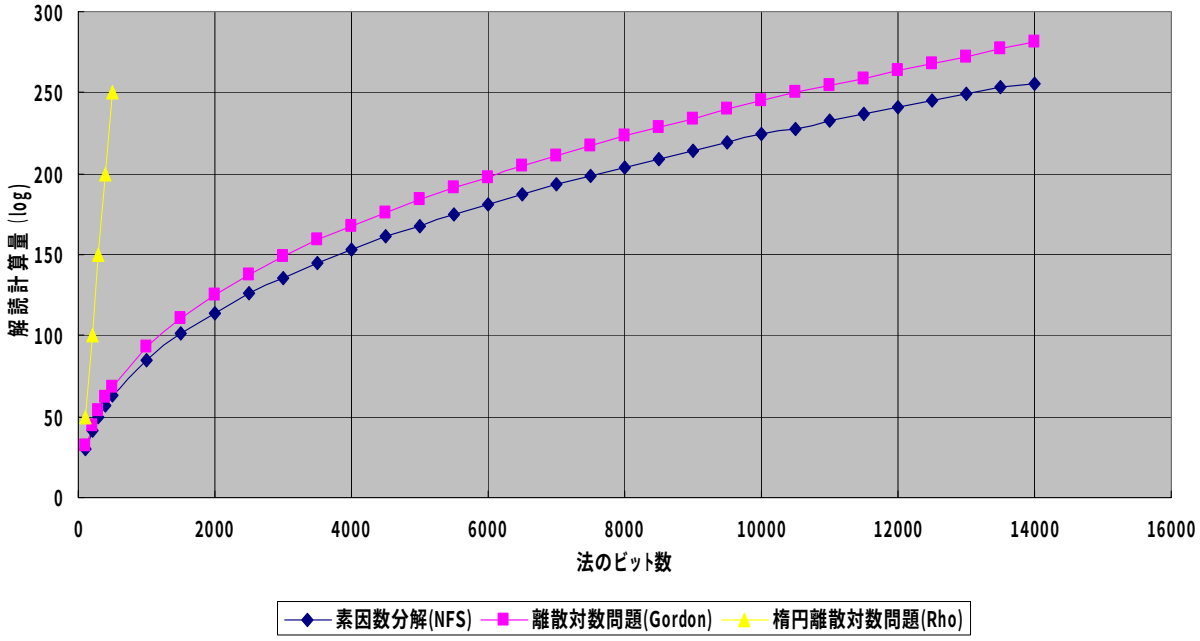
これらはそれぞれのカテゴリーの典型的な暗号方式の特性 (&歴史的理由) に由来する

計算量と実現可能性

暗号プリミティブ (ブロック暗号、ハッシュ関数等)
1回の計算時間を1としたときの総演算量

2^{30}	一瞬	
2^{40}	個人レベル	[かつての米国輸出規制]
$2^{50\text{-}60}$	グループレベル	[DESの鍵サイズ56ビット]
2^{70}	企業レベル	
2^{80}	国家レベル	[RSA1024解読の計算量]

素因数分解および離散対数問題の計算量
(現在知られている最も高速なアルゴリズムによる)



暗号の安全性とは何か (3)

- 「解読可能」は「現実世界で解読可能」を意味するとは限らない
  - Academic attack
    - 暗号の安全性に対して高い水準を要求する
    - この水準は (現実と比べて) しばしば極端に高い
    - このハードルの高さが暗号学の進歩の側面
  - Practical attack
    - 現在あるいは近い将来の技術水準で脅威が現実のものになるかどうかで判断する
    - 理論だけではなく、実装や運用も考慮する

全数探索 / Brute Force Attack (1)

- **全数探索に対する安全性評価の例** (1)
  - 1GHz のクロック 1 サイクルで 1 単位処理 (例えばブロック暗号 1 ブロックの暗号化)
  - そのようなマシンが 100万台並列動作

↓

  - 1秒間に 2^{50} 単位の処理
  - 2^{64} 単位の処理に5時間
  - 2^{80} 単位の処理に34年
  - 2^{128} 単位の処理に10^{16}年 (1京年)

全数探索 / Brute Force Attack (2)

- **全数探索に対する安全性評価の例** (2)
  - 1GHz のクロック 1 サイクルで 1 単位処理 (例えばブロック暗号 1 ブロックの暗号化)
  - そのようなマシンが 100万台並列動作
  - このマシンは 1 年ごとに速度が倍になる

↓

  - 1 年目に 2^{75} 単位/年 の処理
  - 5 年目に 2^{80} 単位/年 の処理
  - 53 年目に 2^{128} 単位/年 の処理
- **安全性評価は技術ﾄﾚﾝﾄﾞ予測に大きく依存**

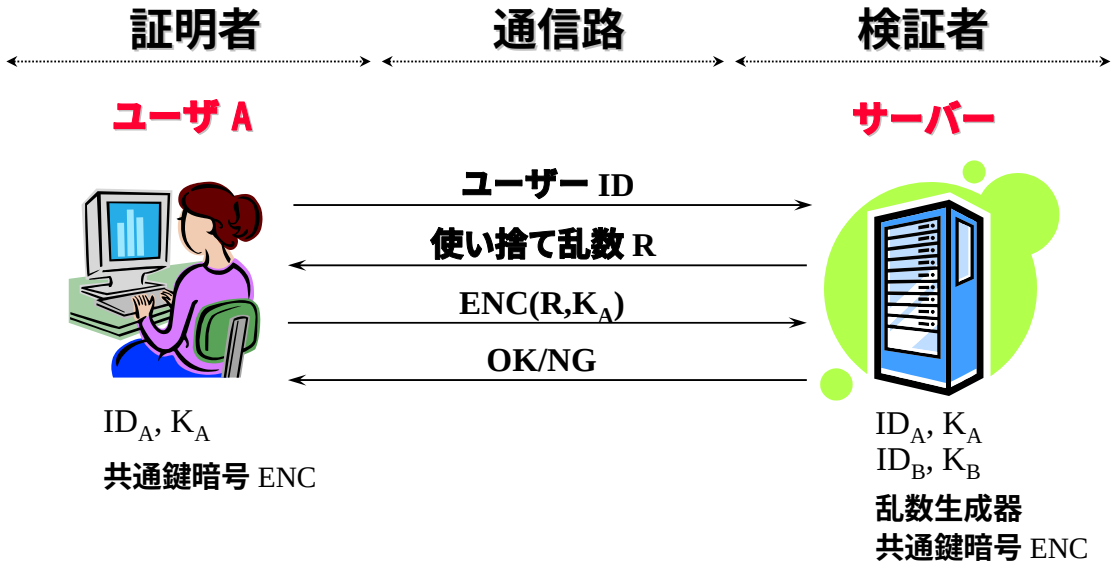
ゲームのルール (1)

- **解読者** (attacker) **に何を許すか** (rules of the game) **の観点から解読** (goal) **の困難さを分類する。**
- **解読者に無限の計算能力とアクセス権を与えるとどんな暗号も「解読可能」。**
- **暗号化鍵は固定されているものとし、解読者が平文/暗号文にアクセスするモデルが一般的。**

ゲームのルール (2)

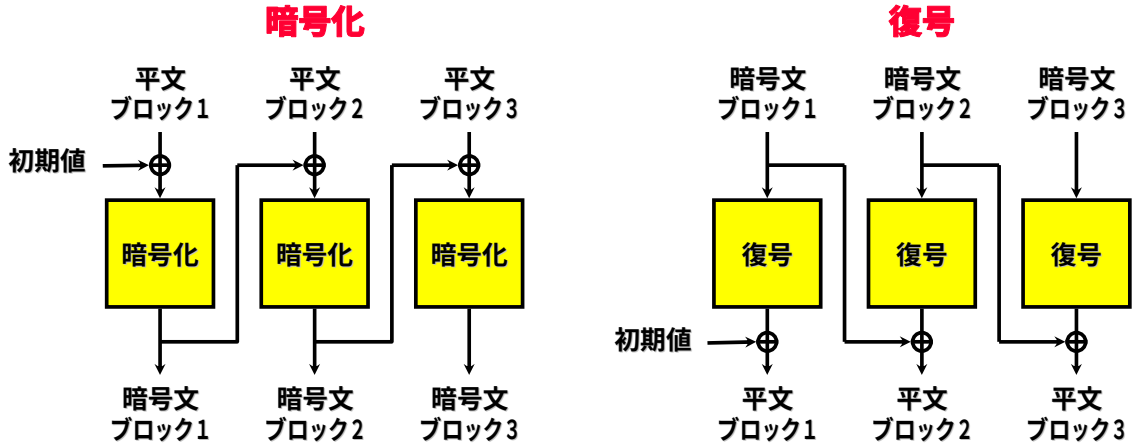
- **暗号文単独攻撃** (ciphertext-only attack)
  - 解読者は暗号文に自由にアクセスできる
  - さらに平文に関する統計情報を既知とする
- **既知平文攻撃** (known plaintext attack)
  - 解読者は平文と暗号文のペアを得ることが出来る
  - ただし平文や暗号文を操作することは出来ない
- **選択平文攻撃** (chosen plaintext attack)
  - 解読者は平文と暗号文のペアを得ることが出来る
  - 解読者は平文を指定することが出来る
- **選択暗号文攻撃** (chosen ciphertext attack)
  - 解読者は平文と暗号文のペアを得ることが出来る
  - 解読者は暗号文を指定することが出来る

共通鍵暗号を用いたユーザ認証

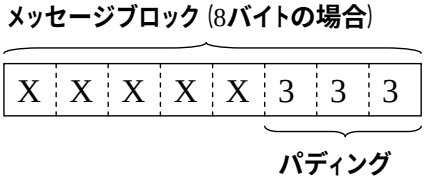


パスワードを直接通信路に流すことなく認証可能
乱数 R は第三者に予測可能であってはならない

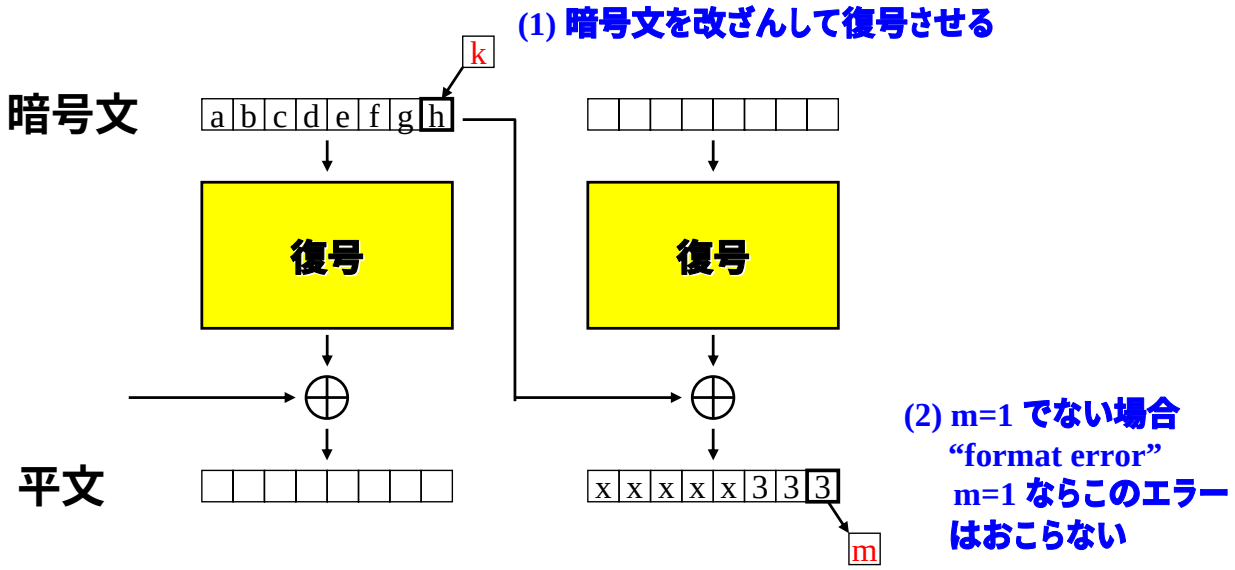
CBCモードとメッセージパディング



PKCS#5 パディング方式



CBCモード暗号文の改ざん攻撃 (1)

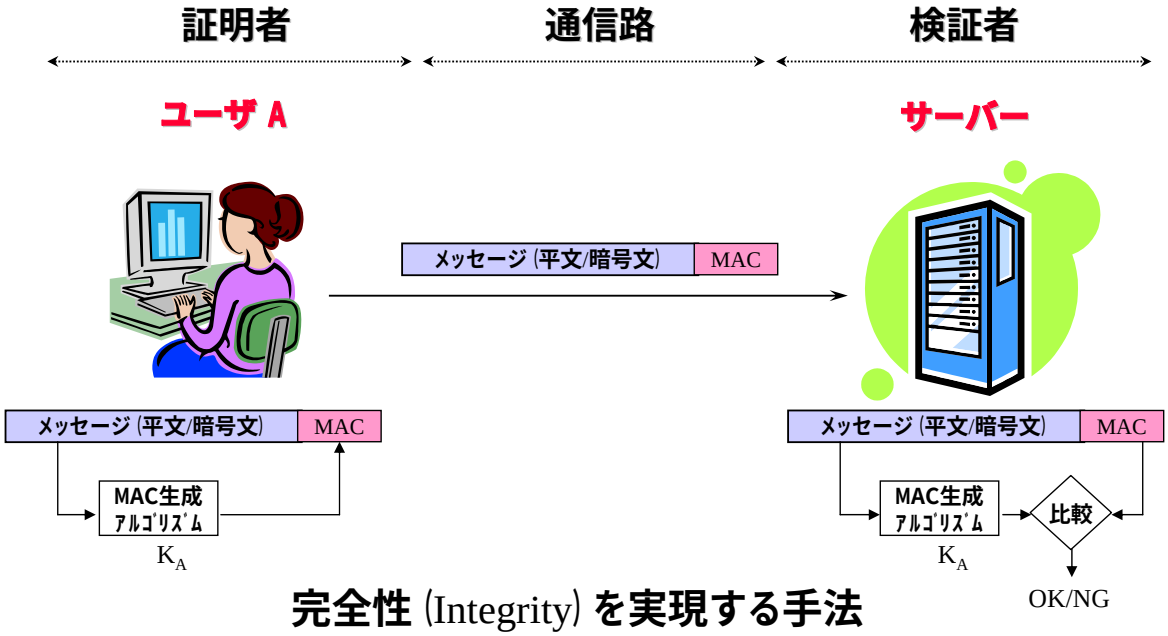


format error をおこさなくなるまで k をいろいろとりかえて試みる
もし format error がおこらなかったら、平文最下位バイトは 1+k+h

CBCモード暗号文の改ざん攻撃 (2)

- CBC モードの実装にかかわる attack
  - By Serge Vaudenay (Eurocrypt 2002)
  - PKCS#5 パディングの復号でエラー処理を利用した一種の選択暗号文攻撃
  - 平均128回の試行ごとに1バイトの平文を導出可能。つまり *n* バイトの平文は平均 *128n* 回の試行で完全解読
  - 復号処理の前に MAC のチェックをすることで回避できる (回避しなければならない)
  - 詳細なエラーメッセージは解読者に利用されることがある

共通鍵暗号を用いたメッセージ認証



完全性 (Integrity) **を実現する手法**
MAC: Message Authentication Code

Special Topic I

ハッシュ関数の安全性をめぐる話題

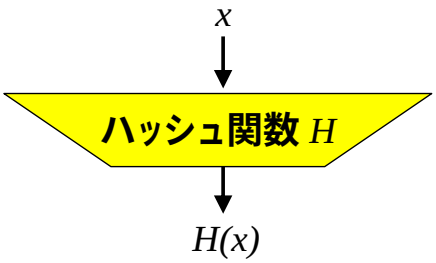
ー Is SHA1 dead? ー

SHA1 問題とは何か

- 2004年中国の研究者 Xiaoyun Wang (王小云) が ハッシュ関数の脆弱性 (Collision) を続々と発表。
  SHA0, RIPEMD, MD4, MD5, HAVAL128, …
- 2005年には Wang は、現米国標準ハッシュ関数 **SHA1** の Collision 探索の計算量を 2^{63} と推定。 本来あるべき SHA1 のセキュリティレベルは 2^{80}。
- SHA1 は世界中で利用されており、影響極めて大。 学会では Wang の主張の検証が大きなトピックに。 NIST は 2011年に新ハッシュ関数の標準化を示唆。

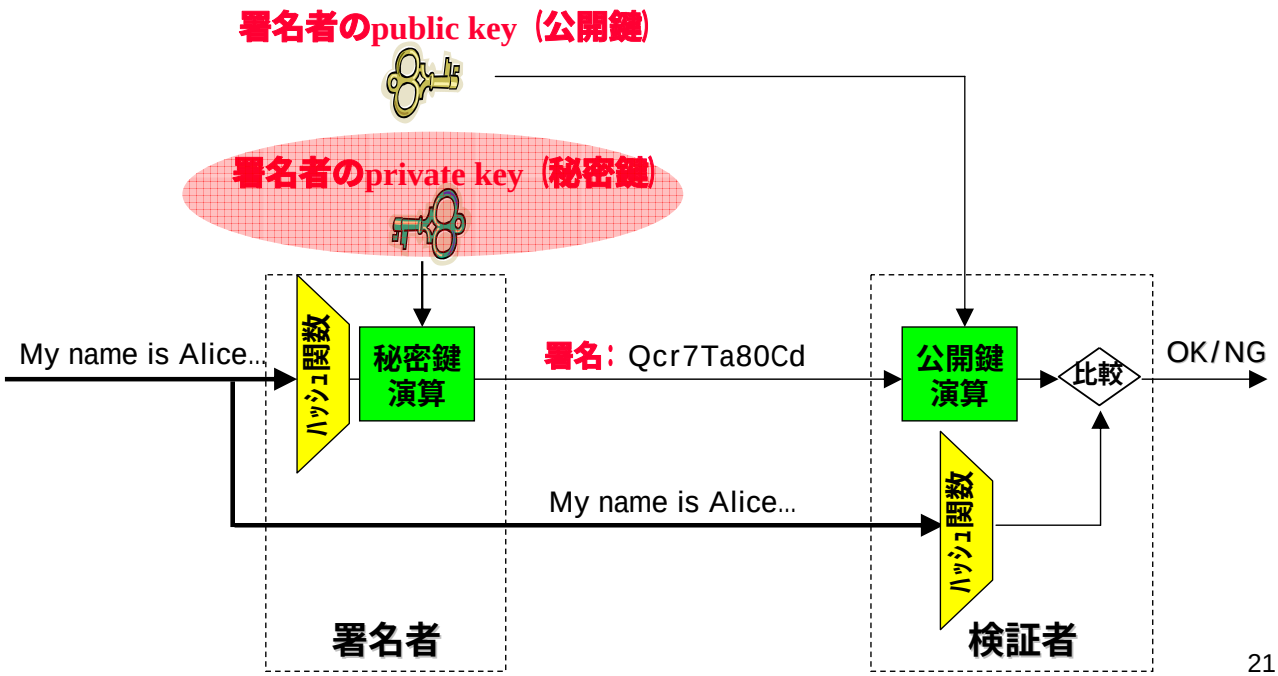
ハッシュ関数 (Hash Function)

- 任意長の入力値を固定長に「圧縮」する関数
- 出力値 (ハッシュ値)から入力値の推測が困難



- 暗号に関する非常に多くの場面で利用される
  - パスワードの暗号化，電子署名，乱数生成 etc
- 暗号学的 (Cryptographic) ハッシュ関数とも言う

ハッシュ関数と電子署名



ハッシュ関数の分類

- ブロック暗号にもとづくハッシュ関数
  - ブロック暗号を繰り返し用いて構成する
  - ISO/IEC 10118-2 で標準化
  - 専用ハッシュ関数より低速
- 専用ハッシュ関数 (dedicated hash functions)
  - ソフトウエアでの高速化を念頭に設計された
  - ISO/IEC 10118-3, FIPS 180-2 で標準化
  - SHA1 が現在のデファクト標準

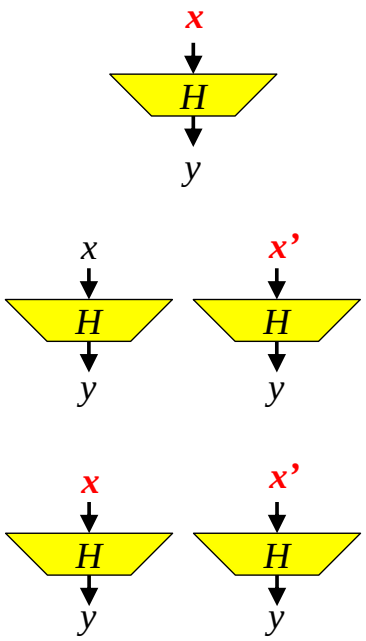
専用ハッシュ関数

	ﾊｯｼｭｻｲｽﾞ	ﾌﾞﾛｯｸｻｲｽﾞ	段数	標準	安全性
MD2	128 bit	128 bit	-	RFC1319	
MD4	128 bit	512 bit	48	RFC1320	
MD5	128 bit	512 bit	64	RFC1321	
SHA	160 bit	512 bit	80	FIPS180-2/ISO18033-3	
SHA1	160 bit	512 bit	80	同上	
SHA256	256 bit	512 bit	64	同上	
SHA512	512 bit	1024 bit	80	同上	

現時点では Collision は見つかりそうにない
Collision が見つかる可能性が高い
Collision がすでに発見された

ハッシュ関数に求められる安全性

- Preimage resistance
  - y から $y=H(x)$ なる x を求めることが困難
- 2nd-preimage resistance
  - $y=H(x)$ なる x, y から $H(x)=H(x')$ なる x' $(\neq x)$ を求めることが困難
- Collision resistance
  - $H(x) = H(x')$ なる x と x' $(x \neq x')$ を求めることが困難



ハッシュ関数の安全性の上限

h : ハッシュ値のビット数 (ハッシュサイズ)

- Preimage resistance
  - 2^h の計算量で preimage を求めることが可能
- 2nd-preimage resistance
  - 2^h の計算量で 2nd preimage を求めることが可能
- Collision resistance
  - $2^{h/2}$ の計算量で collision を求めることが可能 (Birthday Paradox)

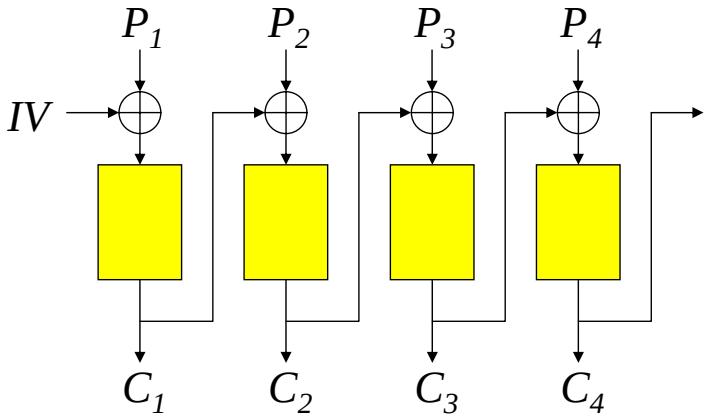
Birthday Paradox

- 無作為に集めた30人の中には、同じ誕生月日の2人が含まれる可能性が高い (およそ70%)
- N 個の集合から、重複を許して無作為に $M=N^{1/2}$ 個程度とると、その中には同じものが含まれている確率が無視できない
- 具体的には、その確率 $p = 1 - exp(-M^2/2N)$ $M=N^{1/2}$ の時 39%, $M=2N^{1/2}$ の時 86%

Birthday Paradox とブロック暗号

- 長らくブロック暗号のブロックサイズは 64 ビットが標準的であった (ex. DES)
- 米国政府新標準ブロック暗号AES ではブロックサイズが 128 ビットに拡大された
- このブロックサイズの変化は birthday paradox を用いたCBC モードの脆弱性の発見による (ciphertext matching attack)

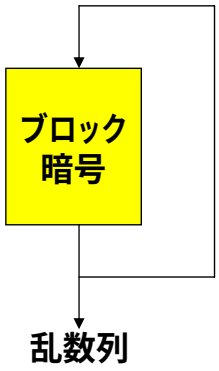
ブロック暗号の CBC モード



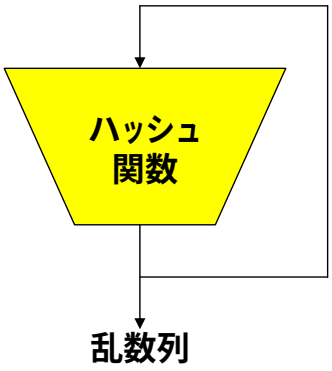
Pi: 平文第 *i* ﾌﾞﾛｯｸ
Ci: 暗号文第 *i* ﾌﾞﾛｯｸ
IV: 初期値
n : ﾌﾞﾛｯｸｻｲｽﾞ (ﾋﾞｯﾄ)

もし $C_i = C_j$ ならば、$P_i + P_j = C_{i-1} + C_{j-1}$ が成り立つ
$2^{n/2}$ ブロック程度平文が与えられると、ある *i* と *j* でこれがおこる可能性が無視できない。 (例) $n=64$ の時、$2^{n/2}$ ブロック = 32GB

乱数生成器としてのハッシュ関数



ブロックサイズがnビットの時
平均周期は 2^{n-1} であることが
知られている

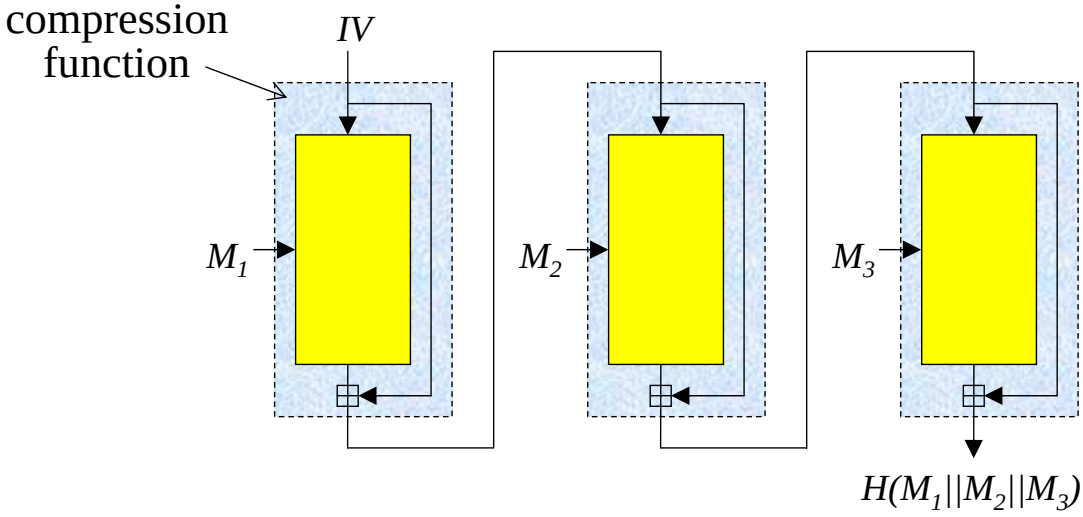


ハッシュサイズがhビットの時
平均周期〜$2^{h/2}$ (birthday paradox)

ハッシュ関数の入力にカウンタ
値を入力すると周期を伸ばせる

専用ハッシュ関数の構造

Merkle-Damgard (MD) Strengthening



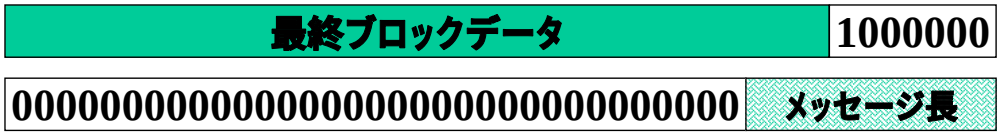
IV (初期値：Initial Vector) は固定値でハッシュ値と同じ長さ
M_i (メッセージブロック) は 512 ビットの場合が多い

メッセージパディング

- メッセージの後ろにデータを付加することで
  その長さをブロックサイズの倍数にする
- パディング＝ストップビット＋0列＋メッセージ長
- ブロックサイズが増加する場合がある

（例1）

（例2）



SHA1: Compression Function

Input/Output: $H^0||H^1||H^2||H^3||H^4$　Message: $M^0||M^1||\ldots||M^{15}$

(1) Message Scheduling

$W^t = M^t \quad (0 \leqq t \leqq 15)$

$W^t = Rol1(W^{t-3} \wedge W^{t-8} \wedge W^{t-14} \wedge W^{t-16}) \quad (16 \leqq t \leqq 79)$

(2) Main Loop

$a=H^0, b=H^1, c=H^2, d=H^3, e=H^4$

For $t=0$ to 79

one round

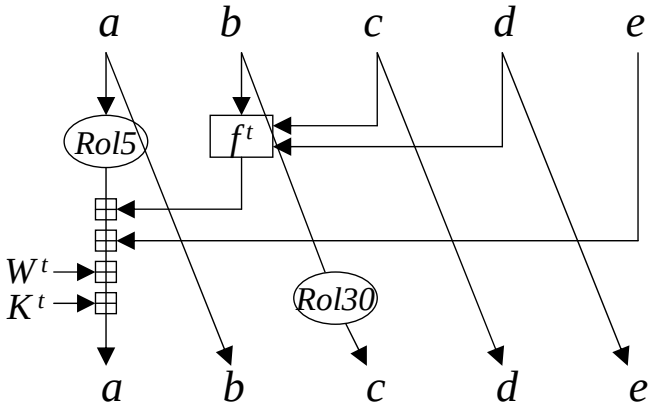
$T=Rol5(a)+f^t(b,c,d)+e+K^t+W^t,$
$e=d, d=c, c=Rol30(b), b=a, a=T$

$H^0=a+H^0, H^1=b+H^1, H^2=c+H^2, H^3=d+H^3, H^4=e+H^4$

SHA1: One Round

$f^t(x, y, z) =$

$$\begin{cases} (x\&y)\hat{}(\sim x\&z) & (0 \leqq t \leqq 19) \\ x\hat{}y\hat{}z & (20 \leqq t \leqq 39) \\ (x\&y)\hat{}(x\&z)\hat{}(y\&z) & (40 \leqq t \leqq 59) \\ x\hat{}y\hat{}z & (60 \leqq t \leqq 79) \end{cases}$$



$(x\&y)\hat{}(\sim x\&z)$: **選択関数** $Ch(x,y,z)$ K^t : **定数**

$(x\&y)\hat{}(x\&z)\hat{}(y\&z)$: **多数決関数** $Maj(x,y,z)$

MD4

- 1990**年** Rivest **によって設計**
- **ﾊｯｼｭｻｲｽﾞ**128**ﾋﾞｯﾄ**, **ﾒｯｾｰｼﾞｻｲｽﾞ**512**ﾋﾞｯﾄ**, 48**段**

- 2^{20}**の計算量で** collision **発見** (Dobbertin '96)
- **手作業で** collision **発見** (Wang**他** '04)
- 2^{56}**メッセージに1つ** preimage **導出可**(Wang**他** '05)

- **このタイプの最初のハッシュ関数**
- **現在は使われていないが、署名検証の目的で必要**

MD5

- 1991年 Rivest によって設計
- ﾊｯｼｭｻｲｽﾞ128ﾋﾞｯﾄ, ﾒｯｾｰｼﾞｻｲｽﾞ512ﾋﾞｯﾄ, 64段

- 別の IV で collision 発見 (Dobbertin '96)
- 2^{39}の計算量で collision 発見 (Wang他 '04)
- 現在 2^{30} (数分) で collision を見つけることができる

- SHA1 の標準化以前のデファクト標準ハッシュ関数
- 現在でも広く用いられている

SHA (SHA0)

- 1993年 NSA によって設計, NIST により規格化
- ﾊｯｼｭｻｲｽﾞ160ﾋﾞｯﾄ, ﾒｯｾｰｼﾞｻｲｽﾞ512ﾋﾞｯﾄ, 80段

- 2^{51}の計算量で collision 導出可 (Joux他 '04)
- 2^{39}の計算量で collision を発見 (Wang他 '05)

- 95年に SHA1 が発表され、SHA0は使用中止に
- 設計原理は現在も非公開 (NSAのポリシー)

SHA1

- 1995年 NSA によって設計, NIST により規格化
- ﾊｯｼｭｻｲｽﾞ160ﾋﾞｯﾄ, ﾒｯｾｰｼﾞｻｲｽﾞ512ﾋﾞｯﾄ, 80段

- 2^{69}の計算量で collision 導出可 (Wang他 '05)
- 2^{63}の計算量で collision 導出可 (Wang他 '05)

- 現在世界のデファクトハッシュ関数

(1) Message Scheduling

$W^t = M^t \quad (0 \leqq t \leqq 15)$

$W^t = \mathbf{Rol1}(W^{t-3} \wedge W^{t-8} \wedge W^{t-14} \wedge W^{t-16}) \quad (16 \leqq t \leqq 79)$

SHA0とSHA1の差

MITSUBISHI 三菱電機 Changes for the Better

SHA1 の現状

- 望ましいセキュリティレベル 2^{80} に対して 2^{63} の計算量で Collision を求めることができる
- 2〜3 年で SHA1 の Collision が発見されると予想されている
- ただし Collision が1つ計算できたとしても現実のアプリケーションに影響はない
- 今から次のハッシュ関数を議論すべき
  - SHA1' or SHA256 or a complete new design?

NISTの声明文

NIST's Policy on Hash Functions March 15, 2006: **The SHA-2 family of hash functions (i.e., SHA-224, SHA-256, SHA-384 and SHA-512) may be used by Federal agencies for all applications using secure hash algorithms.** Federal agencies **should** stop using SHA-1 for digital signatures, digital time stamping and other applications that require collision resistance as soon as practical, and must use the SHA-2 family of hash functions for these applications after 2010. After 2010, Federal agencies may use SHA-1 only for the following applications: hash-based message authentication codes (HMACs); key derivation functions (KDFs); and random number generators (RNGs). Regardless of use, NIST encourages application and protocol designers to use the SHA-2 family of hash functions for all new applications and protocols.

SHA2

- 2002年NISTは新しいハッシュ関数の標準を制定
- SHA1 以外に次の3つのハッシュ関数を登録
  SHA256, SHA384, SHA512
- 2004年にさらに SHA224 をアナウンス
- これらを総称して SHA2 と呼称する

	ﾊｯｼｭｻｲｽﾞ	ﾌﾞﾛｯｸｻｲｽﾞ	段数	処理単位	
SHA224	224 bit	512 bit	64	32 bit	SHA224はSHA256のIVを変え、出力を短縮したもの
SHA256	256 bit				
SHA384	384 bit	1024 bit	80	64 bit	SHA384はSHA512のIVを変え、出力を短縮したもの
SHA512	512 bit				

SHA256: Compression Function

Input/Output: $H^0||H^1||\ldots||H^7$ Message: $M^0||M^1||\ldots||M^{15}$

(1) Message Scheduling

$W^t = M^t \quad (0 \leqq t \leqq 15)$

$W^t = \sigma_1(W^{t-2}) + W^{t-7} + \sigma_0(W^{t-15}) + W^{t-16} \quad (16 \leqq t \leqq 63)$

(2) Main Loop

$a=H^0, b=H^1, \ldots, h=H^7$

For $t=0$ to 63

one round

$T_1 = h + \Sigma_1(e) + Ch(e,f,g) + K^t + W^t,$

$T_2 = \Sigma_0(a) + Maj(a,b,c)$

$h=g, g=f, f=e, e=d+T_1, d=c, c=b, b=a, a=T_1+T_2$

$H^0=a+H^0, H^1=b+H^1, \ldots, H^7=h+H^7$

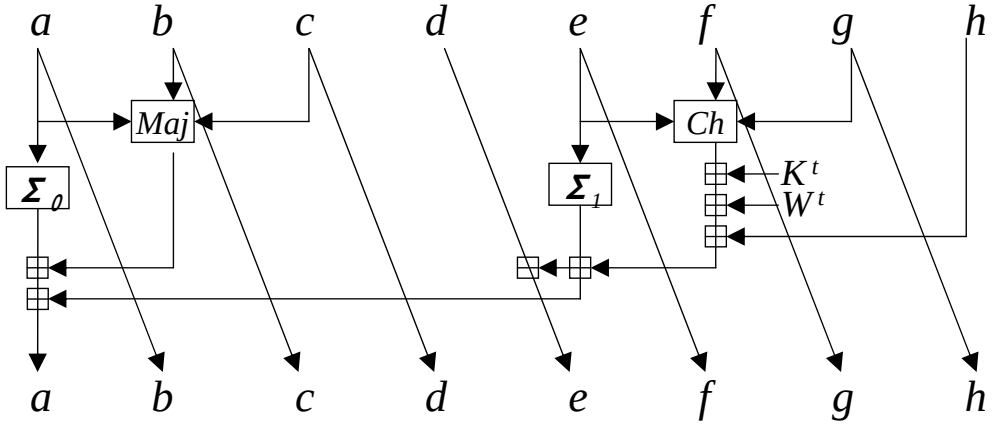
SHA256: One Round

$\sigma_0(x) = Ror2(x) \wedge Ror13(x) \wedge Ror22(x)$

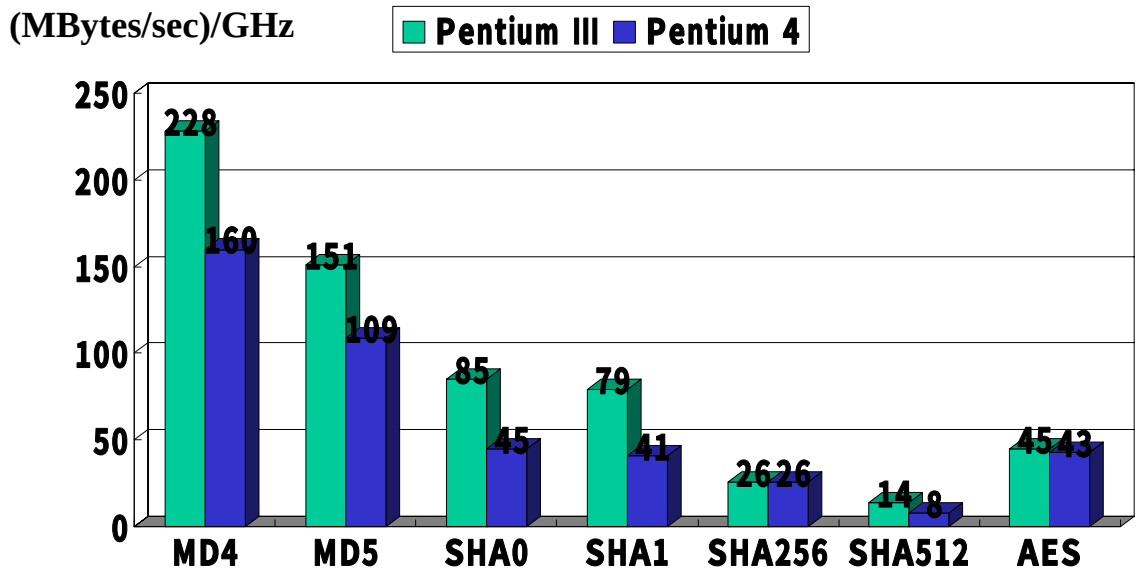
$\sigma_1(x) = Ror6(x) \wedge Ror11(x) \wedge Ror25(x)$

$\Sigma_0(x) = Ror7(x) \wedge Ror18(x) \wedge Shr3(x)$

$\Sigma_1(x) = Ror17(x) \wedge Ror19(x) \wedge Shr10(x)$

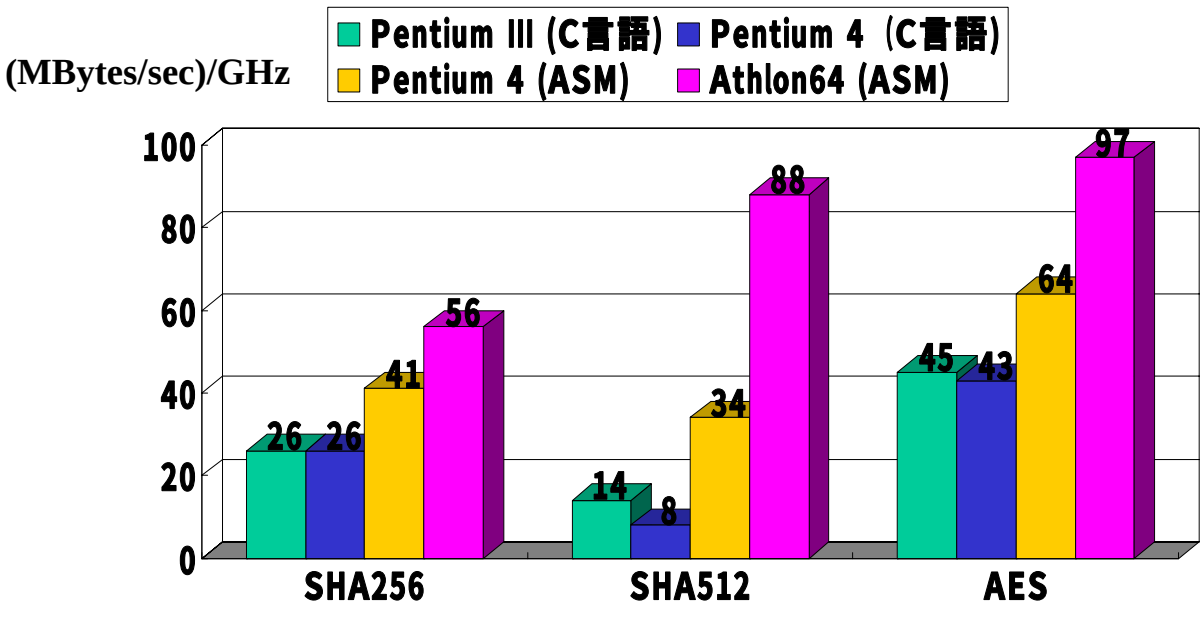


ハッシュ関数の速度 (C言語)



出典: NESSIE report (http://cryptnessie.org/)

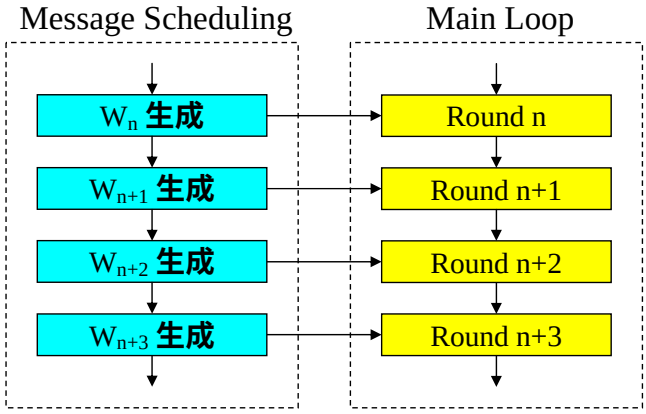
ハッシュ関数の速度 (最適化)



ASM: 著者の書いた 64 ビットアセンブリ言語プログラムでの性能数値

ハッシュ関数の高速化

Message Scheduling と Main Loop をインターリーブ



W_n 生成 → Round n → W_{n+1}生成 → Round n+1 → … の順序で実行

並列処理可能

NISTの次世代ハッシュ関数計画

2008 公募要件検討、公募開始
2009 公募締め切り、第1次評価
2010 第2次評価
2011 次世代ハッシュ関数決定
2012 新 FIPS 登録

AES プロジェクトとほぼ同等のスケジュール
本計画は確定ではなく、詳細は流動的

ハッシュ関数の今後

- ハッシュ関数はセキュリティの根幹をなす技術
- SHA1の継続利用は実用上当面問題ないが…
- 暗号専門家は常に高いセキュリティを求め続ける
- SHA256利用のシステムが増加中
- SHA256は十分安全であると考えられる
- アルゴリズム変更にかかる費用は少なくない
- SHA256にすぐ移行か、2011年を待つかの決断

参考情報

NIST FIPS ドキュメント
http://www.itl.nist.gov/fipspubs/index.htm

NIST Hash ホームページ
http://www.csrc.nist.gov/pki/HashWorkshop/index.html

CRYPTREC ホームページ (日本電子政府暗号プロジェクト)
http://www.cryptrec.jp/

NESSIE ホームページ (欧州暗号評価プロジェクト)
http://cryptonessie.org/

IACR ホームーページ (暗号学会)
http://www.iacr.org/

Special Topic II

PCプロセッサアーキテクチャと暗号

ー How Far Can We Go? ー

暗号屋の宿命

- 暗号処理はしばしば通信のボトルネック
  - 性能がでないと暗号が最初に疑われる！
- 小型（低消費電力）・高速への厳しい要求
  - 性能を稼ぐために安全性を落とせない
- プログラムの巧拙による性能差が大きい
  - 職人芸的アセンブラプログラマ（絶滅寸前）
- 性能は暗号のわかりやすい差別化項目
  - 安全性の差を見える形に示すのは難しい

Block/Hash primitives & Intel processors

RED: lookup tables & logical
BLUE: arithmetic & logical

1976: DES (for hardware)

1987: RC2 (16bit), FEAL (8bit)

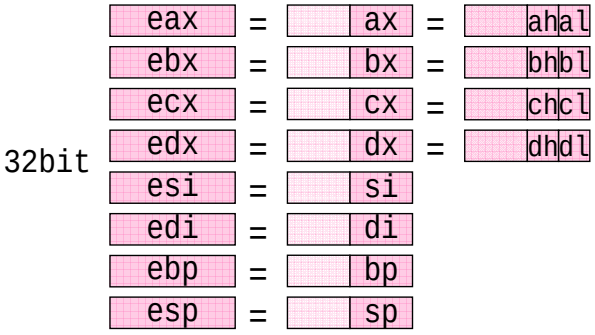
1989: MD2 (16bit)
1990: MD4 (32bit), Multi2 (32bit)
1991: IDEA (16bit)
1992: MD5 (32bit)

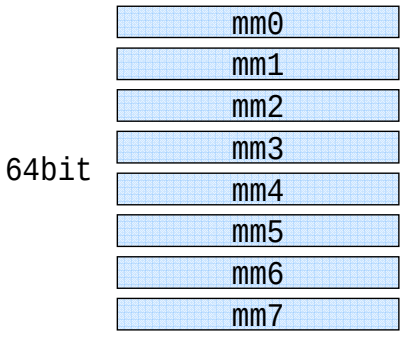
1994: RC5 (32bit)
1995: SHA-1 (32bit)
1996: MISTY1
1998: AES, RC6, Serpent, Mars, Blowfish
2000: Kasumi, Camellia, Whirlpool (64bit)
2002: SHA-2 (32,64bit)
2004: ARIA

70
75
80
85
90
95
00
05

1971: 4004 (4bit,4KB,740KHz) First processor
1974: 8080 (8bit,64KB,2MHz)
1978: 8086 (16bit,1MB,5-10MHz) Segment
1982: 80286 (16bit,16MB,6-12.5MHz) Protect mode
1985: 80386 (32bit,4GB,16-33MHz) Virtual memory
1989: 80486 (25-100MHz) on chip L1 cache
1993: Pentium (60-200MHz) Superscalar
1995: Pentium Pro (150-200MHz)
1997: Pentium II (233-1300MHz) 64-bit MMX
1999: Pentium III (450-1400MHz) SSE
2000: Pentium 4 (-3.4GHz) SSE2 "Northwood"
2003: Pentium M (-2.1GHz)
2004: Pentium 4 (-3.8GHz) SSE3 "Prescott" EM64T
2006: Core (-2.33GHz)
2006: Core2(-2.93GHz) SSE4 EM64T

x86 Architecture

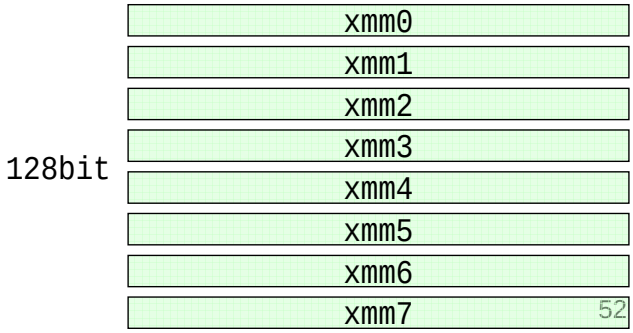




CISC Instruction Set

```
xor   eax, [esi+ebx]
add   12[ebp], al
```

destination source

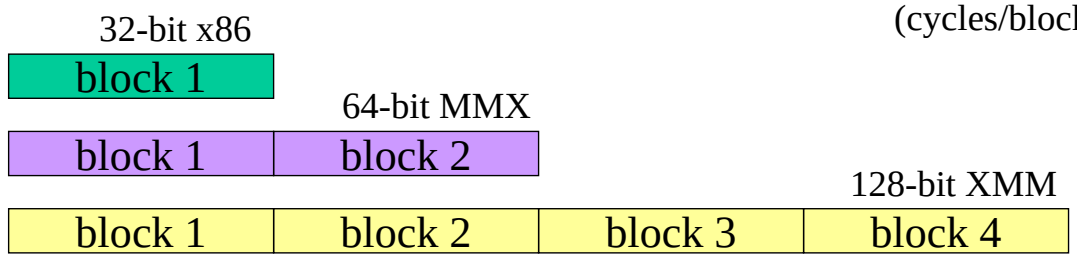


Pentium III & 4: at a glance

Encryption speed of Gladman's Serpent assembly codes optimized for P3

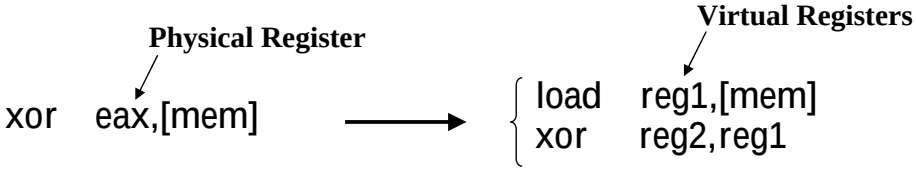
	Pentium III Coppermine	Pentium 4 Northwood	Pentium 4 Prescott
32-bit x86: Straightforward	773	1267	689
64-bit MMX: 2-block parallel	570	1052	1119
128-bit XMM: 4-block parallel	–	656	681

(cycles/block)



Micro-operations (μops)

- Pentium instructions are decomposed into RISC-style simple operations (μops) at the decoding stage
  - Intel has not published exact details on μops
- Programmers cannot direct read/write a code of micro-operations



A Pentium instruction Corresponding μops

Pentium III

- Decoding
  - three decoders: D0, D1, D2
  - complex decoding rules: very often bottleneck in P3
- RAT (register alias table)
  - **3 µops/cycle** : performance limit of P3
  - at most 2 physical registers to virtual registers
- Execution
  - p0,p1: arithmetic/logical p2: read p3,p4: write
  - **out-of-order execution**
  - 1/3 of µops should be memory µops (read/write)

Code Example: SNOW2.0

```
                                                        µops
8B 1D 00000014 R     mov    ebx,[S+20]                 ; 1
0F B6 F2             movzx  esi,dl                     ; 1
C1 C1 08             rol    ecx,8                      ; 1
8B 04 B5 00000848 R  mov    eax,[esi*4+T0]             ; 1
03 DF                add    ebx,edi                    ; 1
0F B6 FE             movzx  edi,dh                     ; 1
33 04 BD 00000C48 R  xor    eax,[edi*4+T1]             ; 2
C1 EA 10             shr    edx,16                     ; 1
0F B6 F9             movzx  edi,cl                     ; 1
33 0D 00000008 R     xor    ecx,[S+8]                  ; 2
0F B6 35 0000002C R  movzx  esi,byte ptr [S+44]        ; 1   27 µops
33 0C BD 00000048 R  xor    ecx,[edi*4+A]              ; 2
8B 3D 0000002C R     mov    edi,[S+44]                 ; 1
C1 EF 08             shr    edi,8                      ; 1
33 0C B5 00000448 R  xor    ecx,[esi*4+AI]             ; 2
33 CF                xor    ecx,edi                    ; 1
0F B6 FA             movzx  edi,dl                     ; 1
89 0D 00000000 R     mov    [S],ecx                    ; 2
33 04 BD 00001048 R  xor    eax,[edi*4+T2]             ; 2
C1 EA 08             shr    edx,8                      ; 1
8B 3C 95 00001448 R  mov    edi,[edx*4+T3]             ; 1
```

MITSUBISHI 三菱電機

Decoding Stage on Pentium III

Changes for the Better

				µops		cycle
8B 1D 00000014 R	mov	ebx,[S+20]	;	1	D0	1
0F B6 F2	movzx	esi,dl	;	1	D1	
C1 C1 08	rol	ecx,8	;	1	D2	
8B 04 B5 00000848 R	mov	eax,[esi*4+T0]	;	1	D0	2
03 DF	add	ebx,edi	;	1	D1	
0F B6 FE	movzx	edi,dh	;	1	D2	
→ 33 04 BD 00000C48 R	xor	eax,[edi*4+T1]	;	2	D0	3
→ C1 EA 10	shr	edx,16	;	1	D1	
0F B6 F9	movzx	edi,cl	;	1	D2	
33 0D 00000008 R	xor	ecx,[S+8]	;	2	D0	4
0F B6 35 0000002C R	movzx	esi,byte ptr [S+44]	;	1	D1	
33 0C BD 00000048 R	xor	ecx,[edi*4+A]	;	2	D0	5
8B 3D 0000002C R	mov	edi,[S+44]	;	1	D1	
C1 EF 08	shr	edi,8	;	1	D2	
33 0C B5 00000448 R	xor	ecx,[esi*4+AI]	;	2	D0	6
33 CF	xor	ecx,edi	;	1	D1	
0F B6 FA	movzx	edi,dl	;	1	D2	
89 0D 00000000 R	mov	[S],ecx	;	2	D0	7
33 04 BD 00001048 R	xor	eax,[edi*4+T2]	;	2	D0	8
C1 EA 08	shr	edx,8	;	1	D1	
8B 3C 95 00001448 R	mov	edi,[edx*4+T3]	;	1	D0	9

MITSUBISHI 三菱電機

Decoding can be a Bottleneck

Changes for the Better

				µops		cycle
8B 1D 00000014 R	mov	ebx,[S+20]	;	1	D0	1
0F B6 F2	movzx	esi,dl	;	1	D1	
C1 C1 08	rol	ecx,8	;	1	D2	
8B 04 B5 00000848 R	mov	eax,[esi*4+T0]	;	1	D0	2
03 DF	add	ebx,edi	;	1	D1	
0F B6 FE	movzx	edi,dh	;	1	D2	
→ C1 EA 10	shr	edx,16	;	1	D0	3
→ 33 04 BD 00000C48 R	xor	eax,[edi*4+T1]	;	2	D0	4
0F B6 F9	movzx	edi,cl	;	1	D1	
33 0D 00000008 R	xor	ecx,[S+8]	;	2	D0	5
0F B6 35 0000002C R	movzx	esi,byte ptr [S+44]	;	1	D0	6
33 0C BD 00000048 R	xor	ecx,[edi*4+A]	;	2	D0	7
8B 3D 0000002C R	mov	edi,[S+44]	;	1	D0	8
C1 EF 08	shr	edi,8	;	1	D1	
33 0C B5 00000448 R	xor	ecx,[esi*4+AI]	;	2	D0	9
33 CF	xor	ecx,edi	;	1	D0	10
0F B6 FA	movzx	edi,dl	;	1	D1	
89 0D 00000000 R	mov	[S],ecx	;	2	D0	11
33 04 BD 00001048 R	xor	eax,[edi*4+T2]	;	2	D0	12
C1 EA 08	shr	edx,8	;	1	D1	
8B 3C 95 00001448 R	mov	edi,[edx*4+T3]	;	1	D0	13

Pentium 4

- Trace Cache（12K entries）
  - Instructions are cached ***after*** being decoded
  - one µop occupies one or two trace cache entries
- Still cannot exceed **3 µops/cycle**
  - can read 6 entries/2 cycles
  - complex rules about trace cache handling
- One-sided focused on high clock frequency
  - Long pipeline stages with many (hidden) stall factors
  - Different units run in different speed
  - Additional cycles if moving different units

Pentium III and 4: a bit more

	Pentium III	Pentium 4 Northwood	Pentium 4 Prescott
Pipeline Stages	10	20	32
L1 data cache	16KB	8KB	16KB
Max. efficiency	3 µops/cycle	3 µops/cycle	2.88 µops/cycle (?)
32-bit load	3 / 1	2 / 1	4 / 1
32-bit add	1 / 0.5	0.5 / 0.25	1 / 0.25
32-bit xor	1 / 0.5	0.5 / 0.5	1 / 0.5
32-bit shift	1 / 1	4 / 1	1 / >0.5
mov ebx,[eax] mov eax,ebx	4 cycles	3 cycles	5 cycles
movq mm0,[eax] mov eax,mm0	4 cycles	13 cycles	18 cycles
Other stall factor	decoding	unit transfer	

(latency/throughput)

How to measure performance

```
xor eax,eax
cpuid
rdtsc
mov CLK1,eax
xor eax,eax
cpuid

Encryption(...,block)

xor eax,eax
cpuid
rdtsc
mov CLK2,eax
xor eax,eax
cpuid
```

```
xor eax,eax
cpuid
rdtsc
mov CLK3,eax
xor eax,eax
cpuid

/* nothing */

xor eax,eax
cpuid
rdtsc
mov CLK4,eax
xor eax,eax
cpuid
```

"Overhead"

((CLK2-CLK1) – (CLK4-CLK3)) / block

Difficulties in Measurement

- Common Implicit Assumptions
  - Should run in a constant time without interruptions
  - Should take more cycles if an interruption takes place
- These assumptions do not hold on Pentium 4 (?)

HT: Hyperthread	Northwood w/o HT	Northwood with HT
Most frequent cycles	632 cycles	636 cycles
Minimum cycles	632 cycles	600 cycles (very rare)

"Overhead" measurement results

Also Prescott Stepping 3 Revision 0 looks unstable

Advanced Encryption Standard

```
Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
   byte  state[4,Nb]

   state = in

   AddRoundKey(state, w[0, Nb-1])                  // See Sec. 5.1.4

   for round = 1 step 1 to Nr-1
      SubBytes(state)                              // See Sec. 5.1.1
      ShiftRows(state)                             // See Sec. 5.1.2
      MixColumns(state)                            // See Sec. 5.1.3
      AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
   end for

   SubBytes(state)
   ShiftRows(state)
   AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])

   out = state
end
```

Figure 5. Pseudo Code for the Cipher.[1]

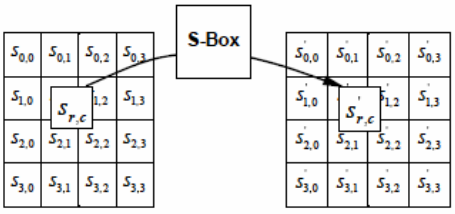


Figure 6. `SubBytes()` applies the S-box to each byte of the State.

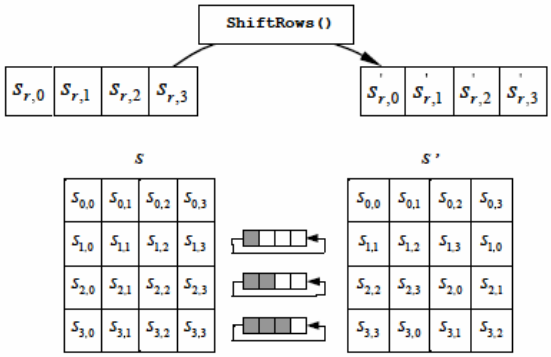


Figure 8. `ShiftRows()` cyclically shifts the last three rows in the State.

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}$$

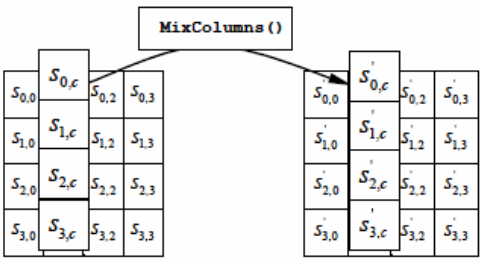


Figure 9. `MixColumns()` operates on the State column-by-colur.....

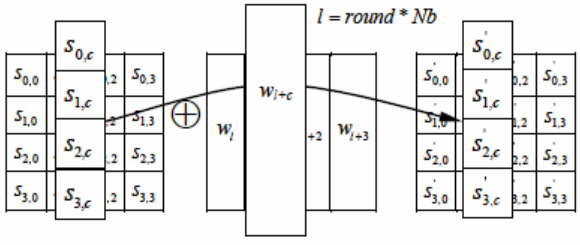
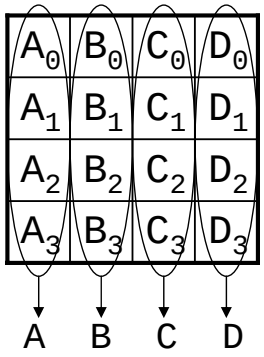


Figure 10. `AddRoundKey()` XORs each column of the State with a word from the key schedule.

One round of AES is simple

ShiftRow+SubBytes+MixColumn

$A' = T0[A_0] \oplus T1[B_1] \oplus T2[C_2] \oplus T3[D_3]$
$B' = T0[B_0] \oplus T1[C_1] \oplus T2[D_2] \oplus T3[A_3]$
$C' = T0[C_0] \oplus T1[D_1] \oplus T2[A_2] \oplus T3[B_3]$
$D' = T0[D_0] \oplus T1[A_1] \oplus T2[B_2] \oplus T3[C_3]$

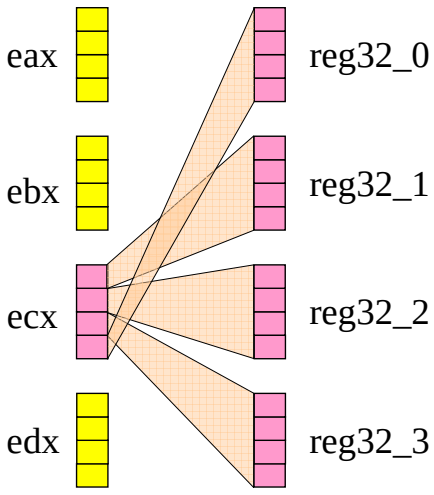


AddRoundKey

$A = A' \oplus KeyA$
$B = B' \oplus KeyB$
$C = C' \oplus KeyC$
$D = D' \oplus KeyD$

A,B,C,D,A',B',C',D':4-byte data
A_i: i-th byte of A
Ti: 1KB table (1byte->4bytes)
Another tables in the final round

AES round function in x86



ShiftRow+SubBytes+MixColumn can be done by a four-time repetition of the following sequence:

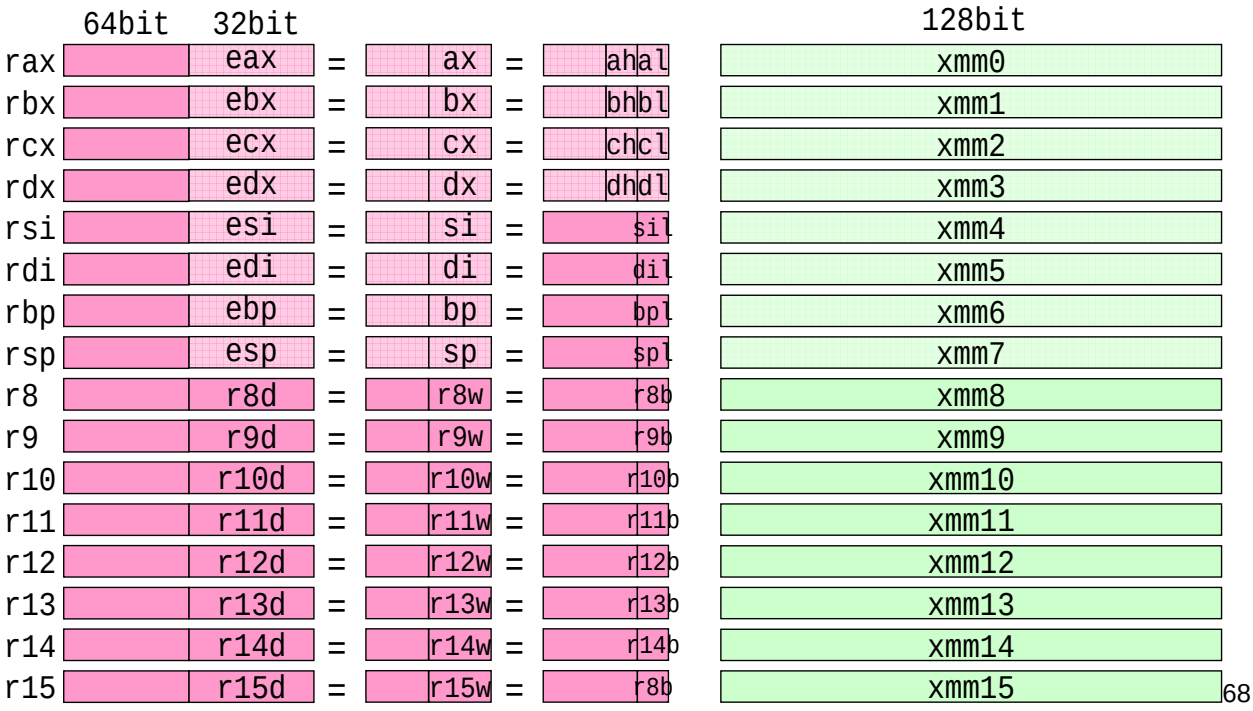
```
movzx    esi,cl
mov/xor  reg32_2,T2[esi*4]
movzx    esi,ch
mov/xor  reg32_1,T1[esi*4]
shr      ecx,16
movzx    esi,cl
mov/xor  reg32_0,T0[esi*4]
movzx    esi,ch
mov/xor  reg32_3,T3[esi*4]
```

Our implementation of AES

	Pentium III	Pentium 4 Northwood	Pentium 4 Prescott
µops / block	596	654	654
cycles / block	232	**251**	**284**
cycles / byte	14.5	**15.7**	**17.8**
µops / cycles	2.57	2.61	2.30

Slow in Prescott probably due to its high load latency

x86 vs. x64: Registers



x64: Better and Worse

(+) more registers, longer registers

(+) most instructions have a 64-bit form

ex) `rol reg32,8 => rol reg64,8`

(-) longer instruction, inefficient decoding

a prefix byte needed for an extended instruction form.

(-) a 64-bit instruction is not always fast

ex) "`shift`" and "`rotate`" on Pentium 4

Pentium 4 vs. Athlon 64

Pentium 4 (Prescott core) *up to 3.8GHz*

(+) long pipeline stages, high clock frequency

(+) instructions are cached after being decoded

(-) poorly documented, never works as Intel claims

Athlon 64 *up to 2.8GHz*

(+) high superscalability (5 uops/cycle)

(+) well documented, less frustrating for programmers

(-) its decoding stage can be a bottleneck

Instruction Latency/Throughput

Processor		Pentium 4 Prescott (EM64T)		Athlon 64 (AMD64)	
Operand Size		32-bit	64-bit	32-bit	64-bit
mov	reg,[mem]	4, 1	4, 1	3, 2	3, 2
mov	reg,reg	1, 2.88	1, 2.88	1, 3	1, 3
add/sub	reg,reg	1, 2.88	1, 2.88	1, 3	1, 3
xor/and/or	reg,reg	1, 2	1, 2	1, 3	1, 3
shr	reg,imm	1, 1.75	7, 1	1, 3	1, 3
shl	reg,imm	1, 1.75	1, 1.75	1, 3	1, 3
ror/rol	reg,imm	1, 1	7, 0.14-1	1, 3	1, 3

latency, throughput

slow 64-bit *right* shifts and 64-bit rotations

Rotate shifts on 64-bit Pentium 4

```
rol  rax,1
rol  rbx,1
rol  rcx,1
rol  rdx,1
rol  rsi,1
rol  rdi,1
rol  rbp,1
```

```
rol  rax,1
xor  r9,r9
rol  rbx,1
xor  r9,r9
rol  rcx,1
xor  r9,r9
rol  rdx,1
xor  r9,r9
rol  rsi,1
xor  r9,r9
rol  rdi,1
xor  r9,r9
rol  rbp,1
```

49 cycles (throughput : 1/7) 7 cycles (throughput : 1)

Some Code Examples

	32 bit	64 bit (1)	64 bit (2)
	xor eax,0[esi+ecx] xor ebx,4[esi+ecx] add ecx,8	xor rax,0[rsi+rcx] xor rbx,8[rsi+rcx] add rcx,16	xor rax,TABLE+0[rcx] xor rbx,TABLE+8[rcx] add rcx,16
Length	10 bytes	13 bytes	**18 bytes**
Pentium 4	2.2 cycles	2.2 cycles	2.2 cycles
Athlon 64	1.0 cycle	1.0 cycle	**1.4 – 1.9 cycles**

	32 bit	64 bit (1)	64 bit (2)
	movzx ecx,al xor ebx,[esi+ecx*4] shr eax,8	movzx rcx,al xor rbx,[rsi+rcx*8] **shr rax,8**	movzx rcx,al xor rbx,TABLE[rcx*8] **shr rax,8**
Length	9 bytes	12 bytes	16 bytes
Pentium 4	1.7 cycles	**7.0 cycles**	**7.0 cycles**
Athlon 64	1.0 cycle	1.0 cycle	1.0 cycle

Performance of AES on x64 Processors

- The structure of AES is optimized for 32-bit processors.
- Free from “register starvation” due to 16 general registers.

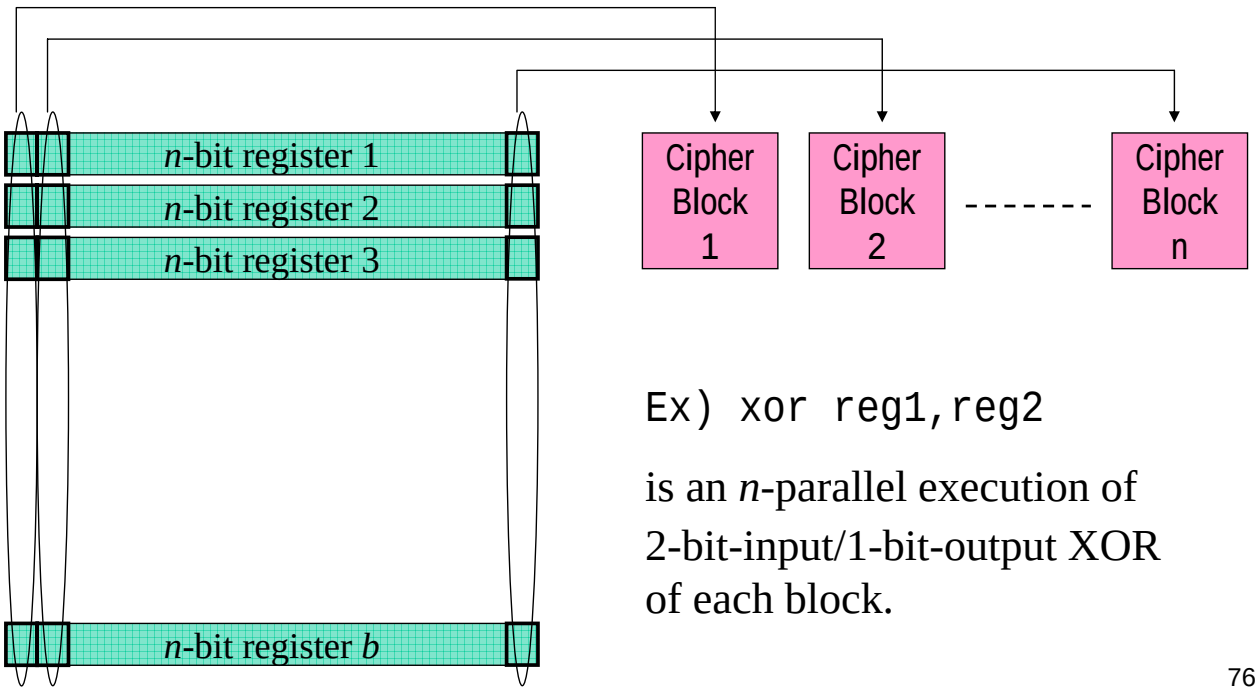
Performance of AES (128-bit key) on Athlon64/Pentium 4

	AES		
Processors	Athlon 64 64-bit	Pentium 4 64-bit	Pentium 4 32-bit
cycles/block	170	256	284
instructions/cycle	2.74	1.81	-
uops/cycle	3.53	2.34	-

Bitslice Implementation of Block Ciphers

- Introduced by Biham (FSE'97)
- *n*-block parallel execution using *n*-bit registers
- *1* software instruction = *n* simple hardware gates
  - `AND, OR, XOR, NOT...`
- Very efficient if
  - registers are long
  - registers are many
  - the target algorithm is small in hardware
- Protection against cache timing attack

Principle of Bitslice Implementation



`Ex) xor reg1,reg2`

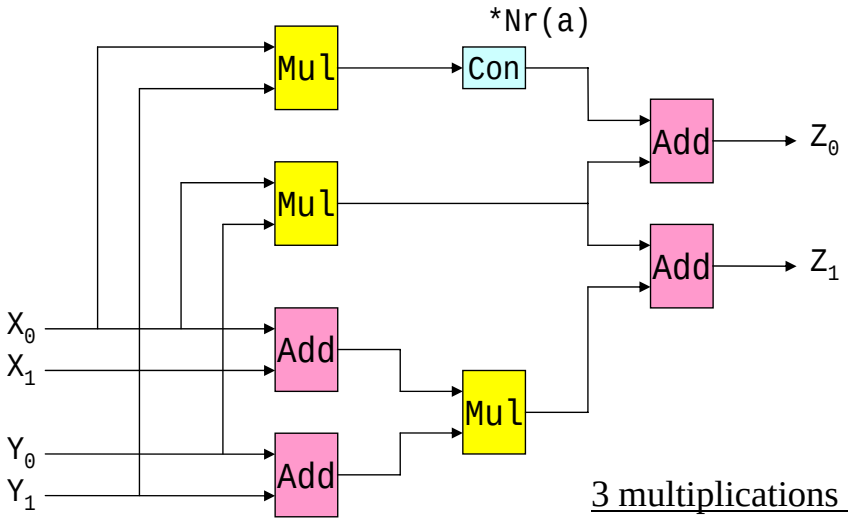
is an *n*-parallel execution of 2-bit-input/1-bit-output XOR of each block.

Bitslice and S-box

- Many recent block ciphers have adopted an 8x8 S-box (a lookup table), linearly equivalent to an inversion over GF(2^8).
  - AES, Camellia, SNOW2.0, ARIA etc
- An inversion over GF(2^8) is strong against differential/linear attacks (actually best known), but can be weak against cache timing attacks.
- The bitslice implementation can compute an inversion over GF(2^8) without a table lookup.

Multiplication over GF(2^{2n}) using GF(2^n)

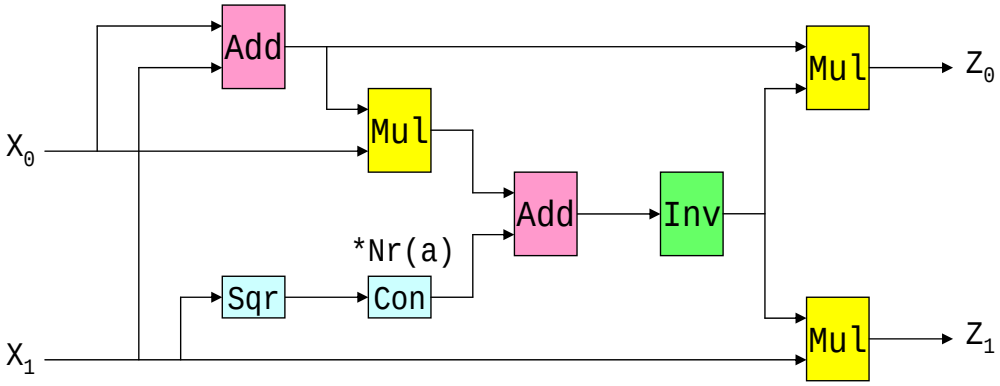
Basis of GF(2^{2n})/GF(2^n): (1, a)



3 multiplications over GF(2^n)

$Z_0+Z_1a = (X_0+X_1a)(Y_0+Y_1a)$ where $Tr(a)=1$

Inversion over GF(2^{2n}) using GF(2^n)



$Z_0+Z_1a = 1/(X_0+X_1a)$ where $Tr(a)=1$

Implementation Results

The Full AES S-box

Basis Change (before inversion)	Inversion	Basis Change (after inversion)	Constant XOR	Total
12	177	16	(4)	205 (209)

Performance of Bitsliced AES/Camellia on Athlon64/Pentium 4

	AES		Camellia	
Processors	Athlon 64	Pentium 4	Athlon 64	Pentium 4
cycles/block	250	418	243	415
instructions/cycle	2.75	1.66	2.74	1.61
uops/cycle	3.20	1.93	2.99	1.75

Concluding Remarks

- A combination of lookup tables and logical operations is suitable for both software and hardware.
- Understanding hardware is important in doing software.
- Pentium 4 looks a dead end of processor design
  - The long pipeline leads to an overheating problem
  - AMD Athlon64 very often runs faster than Pentium 4
- Parallel encryption will be increasingly important
- Intel's new 'Core' processors go back to Pentium III
  - Bitsliced ciphers can be much faster on Core2

References

M. Matsui, S. Fukuda: *"How to Maximize Software Performance of Symmetric Primitives on Pentium III and 4 Processors"*, Fast Software Encryption Workshop 2005

M. Matsui: *"How Far Can We Go on the x64 Processors?"* , FSE2006, Fast Software Encryption Workshop 2006

A. Fog: *Software optimization resources*, http://www.agner.org/optimize/

Intel resource: http://developer.intel.com/

AMD resource: http://www.amd.com/us-en/Processors/TechnicalResources/0,,30_182_739,00.html