Pseudorandom Functions and Chosen Plaintext Attacks

Cryptography and Protocols

Andrei Bulatov

Two Problems of PRG-Based Encryption Schemes

- Single key – multiple messages

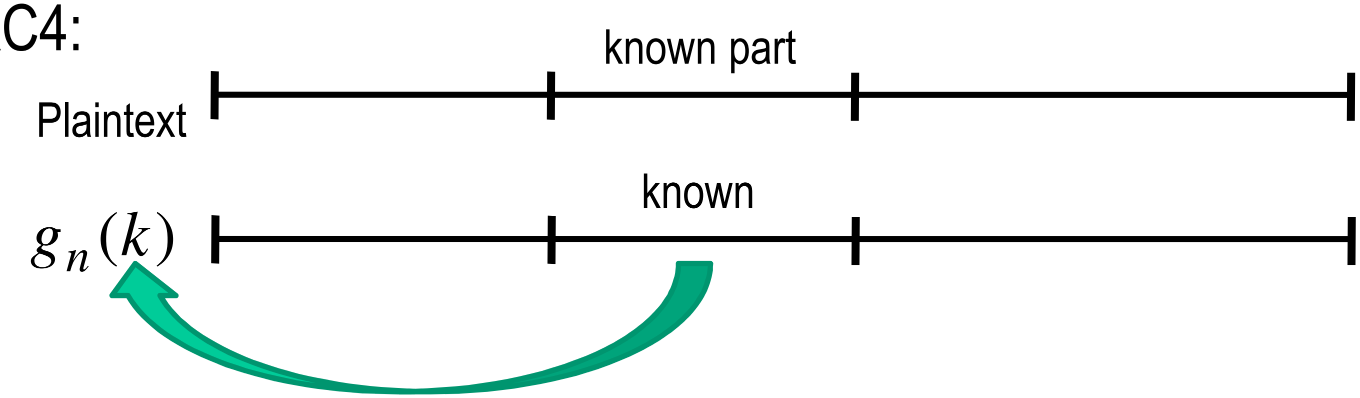
  All the theoretically analyzed schemes aim to send only one message, while in practice we need to send multiple messages.

  Ad hoc practical schemes may be susceptible to collision, reply, and all kinds of unknown attacks

- Chosen plaintext attacks

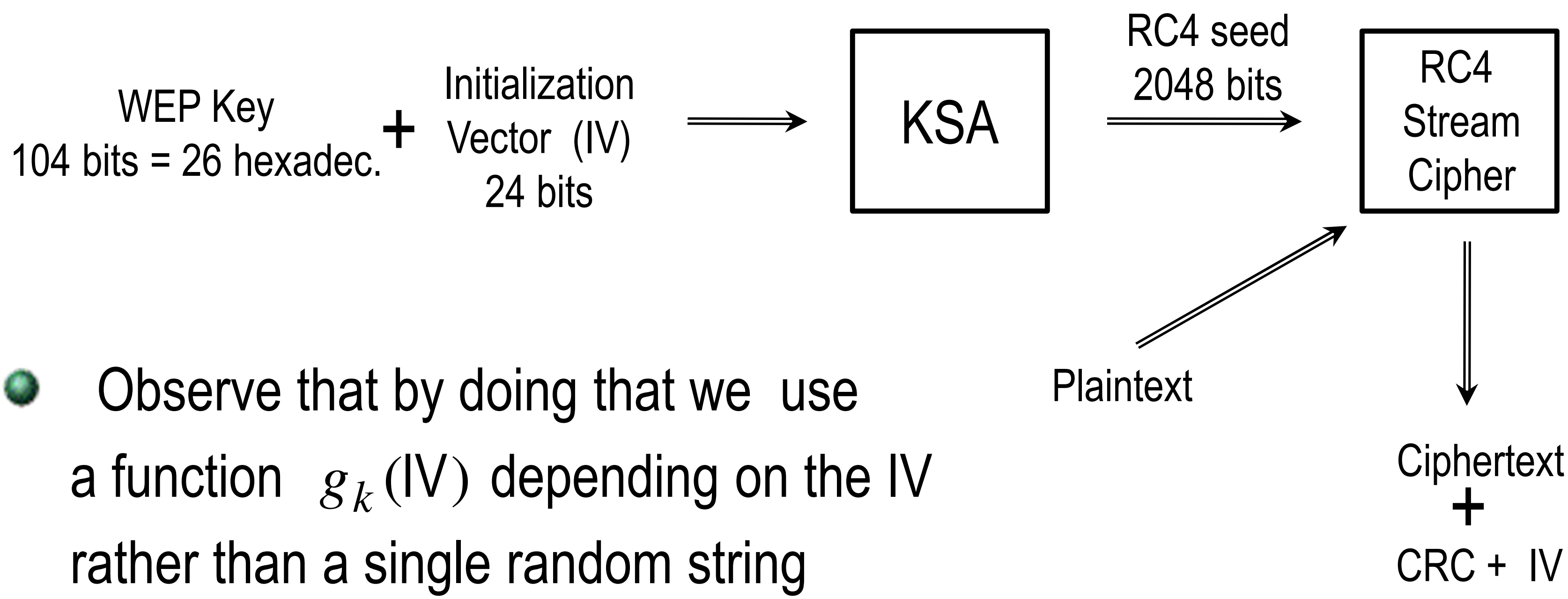
  We assumed that Eve cannot choose what plaintext to encrypt. However sometimes it is possible and may lead to consequences.

  For RC4:



Solutions to the Multiple Messages Problem

- Add a variable part to the key that is safe to send in clear



- Observe that by doing that we use a function $g_k(\mathrm{IV})$ depending on the IV rather than a single random string
- This is the idea behind pseudorandom functions

Random Functions

- A random function $f_n : \{0,1\}^n \to \{0,1\}^n$ is built as follows: for each of the 2^n inputs we choose at random an n-bit string. Thus, f_n is $n \cdot 2^n$-bit string

- Let $F = \{f_s\}_{s \in \{0,1\}^*}$ be a collection of functions such that $f_s : \{0,1\}^{m(|s|)} \to \{0,1\}^{m(|s|)}$ This collection is efficiently computable if the mapping $s, x \mapsto f_s(x)$ is computable in polynomial time

Pseudorandom Functions: Games

- Let $F = \{f_s\}_{s\in\{0,1\}^*}$ be a collection of functions, T a function, Eve an algorithm
- **Game 1**

  - s is chosen uniformly at random from $\{0,1\}^n$

  - Eve gets black-box access to the function f_s for as long as it wants, but no more than T(n)

  - Eve outputs a bit v
- **Game 2**

  - a random function is chosen $f:\{0,1\}^n \to \{0,1\}^n$

  - Eve gets black-box access to the function f for as long as it wants, but no more than T(n)

  - Eve outputs a bit v

Pseudorandom Functions: Definition

- Let $F = \{f_s\}_{s \in \{0,1\}^*}$ be a collection of functions, T, ε a pair of functions
- F is said to be (T,ε)-pseudorandom if it is efficient and for any algorithm Eve of time complexity at most T

$$|\Pr[\text{Eve}(\text{Game 1}) = 1] - \Pr[\text{Eve}(\text{Game 2}) = 1]| < \varepsilon(n)$$

Chosen Plaintext Attacks

- Game of attacking a SES (K,E,D):

  - Eve chooses P_1, P_2

  - Alice chooses a random key $k \in \{0,1\}^n$ and i ∈ {1,2} and sends $E_k(P_i)$

  - for as long as Eve wants (but no more than her running time) she gets access to E_k as a black box: she chooses P and sees $E_k(P)$

  - Eve comes up with a guess j. She is successful if j = i

CPA Security

- A SES (K,E,D) is (T,ε)-CPA secure if for any algorithm Eve with running time at most T in the game from the previous slide

$$\Pr[j = i] < \tfrac{1}{2} + \varepsilon(n)$$

- (K,E,D) is said to be CPA secure if it is (T,ε)-CPA secure for a some superpolynomial pair

SES Based on PRF

- Let $F = \{f_s\}_{s \in \{0,1\}^*}$ be a PRF.
- A SES (K,E,D) is defined as

  K – chooses key k uniformly at random from $\{0,1\}^n$

  E – chooses r at random from $\{0,1\}^n$

  and sends $< r, f_k(r) \oplus P >$

  D – computes $f_k(r) \oplus C$ to decrypt $< r, C >$

- **Theorem**

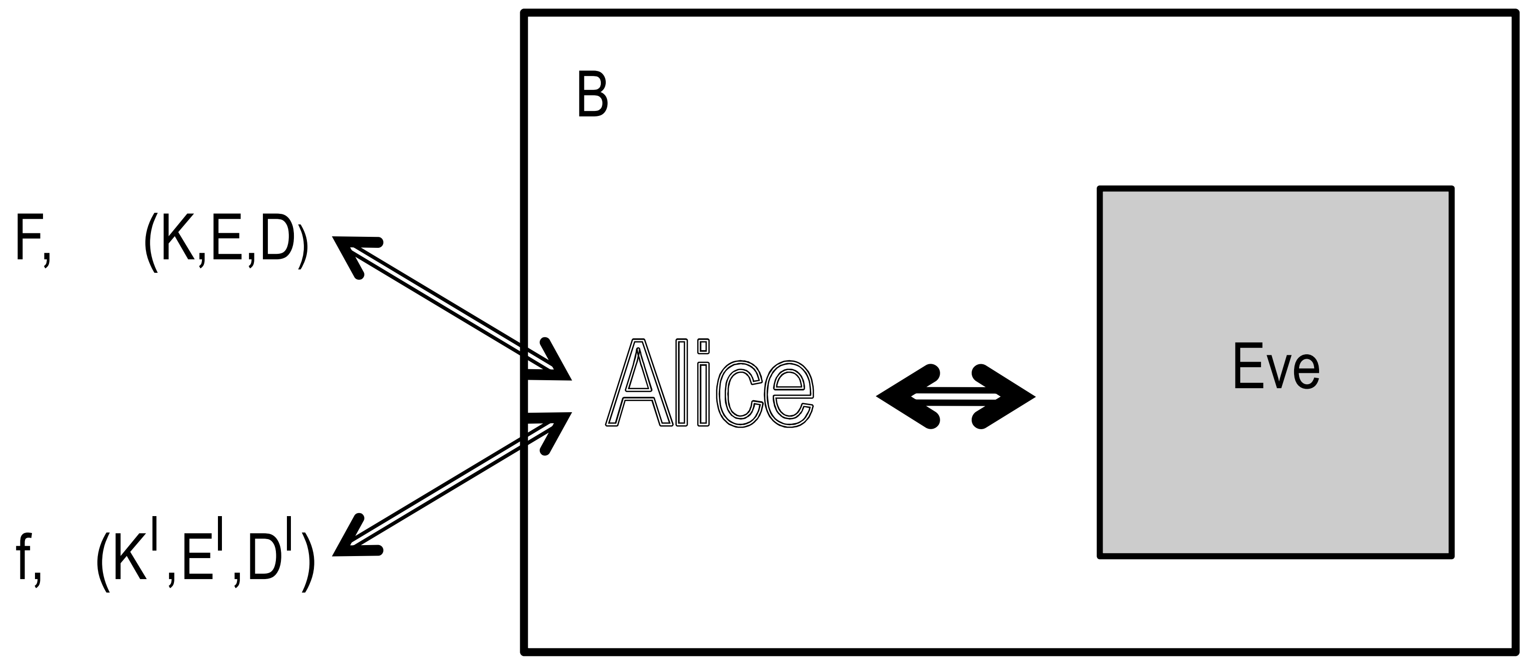
  Let (K,E,D) be the SES based on a (T,ε)-collection of pseudorandom functions F. If $T << 2^n$ then (K,E,D) is (T,ε)-CPA secure

CPA Security (cntd)

- **Proof**

  Suppose there is Eve that breaks (K,E,D). We construct a distinguisher B for F.

  B uses Eve as subroutine



CPA Security (cntd)

- Distinguisher B has access to either F or a random function f. Denote the function it has access to by G
- Distinguisher B works as follows
  - generate a random bit b
  - start Eve
  - when Eve generates a query P_1, P_2 to Alice do:
    - generate a random key k and string r, $k, r \in \{0,1\}^n$
    - encrypt $C = P_{b+1} \oplus G_k(r)$ and return $< r, C >$ to Eve
  - when Eve generates a CPA query P do:
    - generate a random string r
    - encrypt $C = P \oplus G_k(r)$ and return $< r, C >$ to Eve
  - when Eve returns answer b’: if b’ = b output 1, otherwise 0

CPA Security (cntd)

- If Game 1 is played, that is, real (K,E,D) with F

  B outputs 1 $\Leftrightarrow$ Eve is successful with (K,E,D)

  $$\Pr[B(\text{Game 1}) = 1] = \Pr[\text{Eve wins with } (K, E, D)] \geq \tfrac{1}{2} + \varepsilon$$

- If Game 2 is played
- **Lemma**.

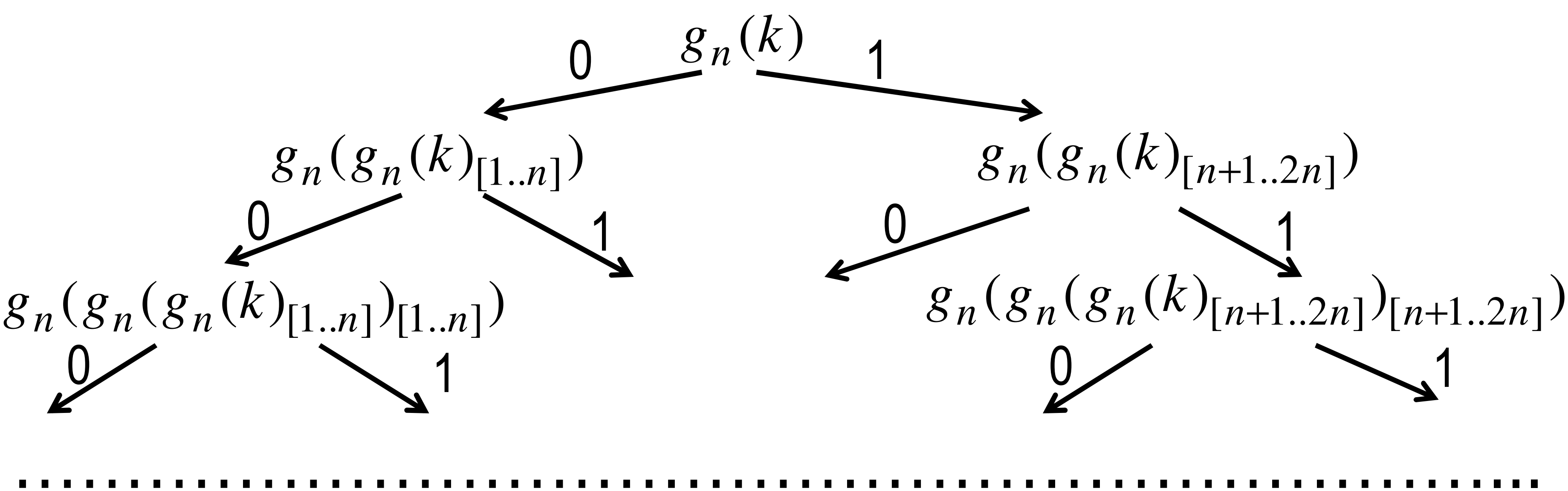
  The probability that an adversary that makes less than $2^{n/4}$ queries in a CPA attack on (K^I,E^I,D^I) guesses P_{b+1} is less than $\tfrac{1}{2} + 2^{-n/2}$

- The result follows:

$$|\Pr[B(\text{Game 1}) = 1] - \Pr[B(\text{Game 2}) = 1| > \varepsilon - 2^{-n/2}$$

PRF from PRG

- Suppose we have a pseudorandom generator $\{g_n\}$,
  $g_n : \{0,1\}^n \to \{0,1\}^{2n}$ Can we construct a PRF?
  Generating a $n \cdot 2^n$-bit string does not work
- For a seed k construct a tree



$f_k(00...0)$ $f_k(00...01)$ $f_k(11...1)$

PRF from PRG (cntd)

- Formally. Let $g_n^0(t) = g_n(t)_{[1..n]}$ and $g_n^1(t) = g_n(t)_{[n+1..2n]}$
  Then
  $$f_k(x_1 x_2 \ldots x_n) = g_n^{x_n}(g_n^{x_{n-1}}(\ldots g_n^{x_1}(k) \ldots))$$

- **Theorem**

  If $\{g_n\}$ is a (T,ε)-pseudorandom generator for a superpolynomial pair T, ε, then $\{f_k\}$ is a (T,ε)-pseudorandom collection of function

Candidates for PRF: Naor-Reingold-Rosen

- Uses the same idea as Blum-Blum-Shub
- For each n let

  N be an n-bit number

  $\bar{a} = (a_{10}, a_{11}, a_{20}, a_{21}, \dots, a_{n0}, a_{n1})$ be a sequence of 2n numbers from {1, …, N -1}

  $g \equiv b^2 \pmod N$ for some b

  r be an n-bit string

- For any n-bit sequence $x = x_1 \dots x_n$ we set

$$h_{N,\bar{a},g}(x) = g^A \pmod N \quad \text{where} \quad A = \prod_{i=1}^{n} a_{ix_i}$$

Candidates for PRF: Naor-Reingold-Rosen (cntd)

- Then we use $h_{N,\bar{a},g}(x)$ as a seed for a BBS-like process

  i := 0

  X := $h_{N,\bar{a},g}(x)$

  while i ≤ n

  X := X · X (mod N)

  i := i + 1

  output $X_1 \cdot r_1 \oplus \cdots \oplus X_n \cdot r_n \pmod 2$

  endwhile

- **Theorem**

  If the Factoring Assumption (that factoring an integer cannot be done in poly time) holds then the collection of functions above is an efficiently computable PRF

Candidates for PRF: Hash Function

- Hash functions are used to produce a bit string of fixed size from a string of any size
- Main requirement: It is hard to find a collision, a pair of inputs that produce the same result
- Many hash functions, such as MD5 (Message Digest), see next slide, also use an initialization vector. IV usually fixed.

MD5

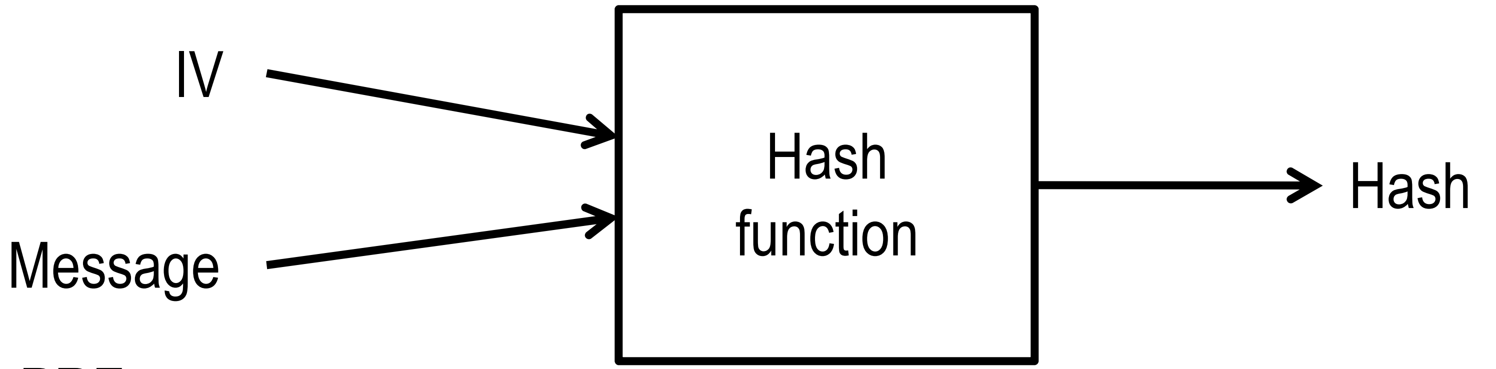
```
//Note: All variables are unsigned 32 bits and wrap modulo 2^32 when calculating
var int[64] r, k /

/r specifies the per-round shift amounts
r[ 0..15] := {7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22}
r[16..31] := {5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20}
r[32..47] := {4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23}
r[48..63] := {6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21} /

/Use binary integer part of the sines of integers (Radians) as constants:
for i from 0 to 63
    k[i] := floor(abs(sin(i + 1)) × (2 pow 32))

//Initialize variables:
var int h0 := 0x01234567
var int h1 := 0x89ABCDEF
var int h2 := 0xFEDCBA98
var int h3 := 0x76543210 /
```

IV

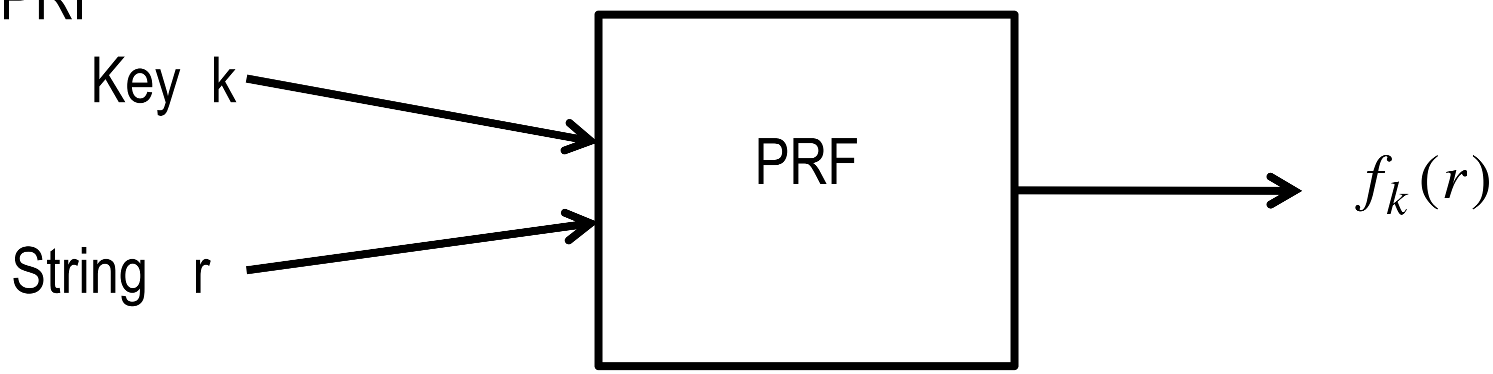
```
/Pre-processing:
append "1" bit to message
append "0" bits until message length in bits ≡ 448 (mod 512)
append bit /* bit, not byte */ length of unpadded message as
    64-bit little-endian integer to message /

/Process the message in successive 512-bit chunks:
for each 512-bit chunk of message
   break chunk into sixteen 32-bit little-endian words w[i], 0 ≤ i ≤ 15
   //Initialize hash value for this chunk:
   var int a := h0
   var int b := h1
   var int c := h2
   var int d := h3

//Main loop:
   for i from 0 to 63
      if 0 ≤ i ≤ 15 then
         f := (b and c) or ((not b) and d)
         g := i
      else if 16 ≤ i ≤ 31
         f := (d and b) or ((not d) and c)
         g := (5×i + 1) mod 16
      else if 32 ≤ i ≤ 47
         f := b xor c xor d
         g := (3×i + 5) mod 16
      else if 48 ≤ i ≤ 63
         f := c xor (b or (not d))
         g := (7×i) mod 16

      temp := d
      d := c
      c := b
      b := b + leftrotate((a + f + k[i] + w[g]) , r[i])
      a := temp
   //Add this chunk's hash to result so far:
   h0 := h0 + a
   h1 := h1 + b
   h2 := h2 + c
   h3 := h3 + d

var int digest := h0 append h1 append h2 append h3 //(expressed as little-endian)
```

PRF from Hash Function

- Hash function



- PRF



- **Theorem**

  If the hash function is good, then $\{f_k\}$ is a PRF

WPA

- Recall WPA