2026/07/08, 2026 보안 기초 세미나

암호학과 네트워크 보안

-12장 암호학적 해시함수-

송 민 희(minhee@pel.sejong.ac.kr)

세종대학교 프로토콜공학연구실

목 차

- 개요
- SHA-512
- WHIRLPOOL

목 차

개요

- 암호학적 해시함수(Cryptographic Hash Function)
  - 정의
    - 임의 길이의 입력 데이터를 받아 고정 길이의 해시 값으로 변환하며, 3가지의 저항성 기준을 충족하는 해시함수
  - 저항성 기준
    - 역상 저항성(Preimage Resistance)
      - 어떤 메시지 M에 대한 해시 값 $h(M)$만을 가지고 메시지 M을 찾기 어렵게 하는 성질
    - 제 2 역상 저항성(Second Preimage Resistance)
      - 어떤 메시지 M과 해시 값 $h(M)$이 있을 때, $h(M) = h(M')$을 만족하는 M'을 찾기 어렵게 하는 성질
    - 충돌 저항성(Collision Resistance)
      - 서로 다른 두 메시지 M, M'에 대해서 $h(M) = h(M')$을 만족하는 임의의 2개의 입력 값(M, M')을 구하지 못하는 성질

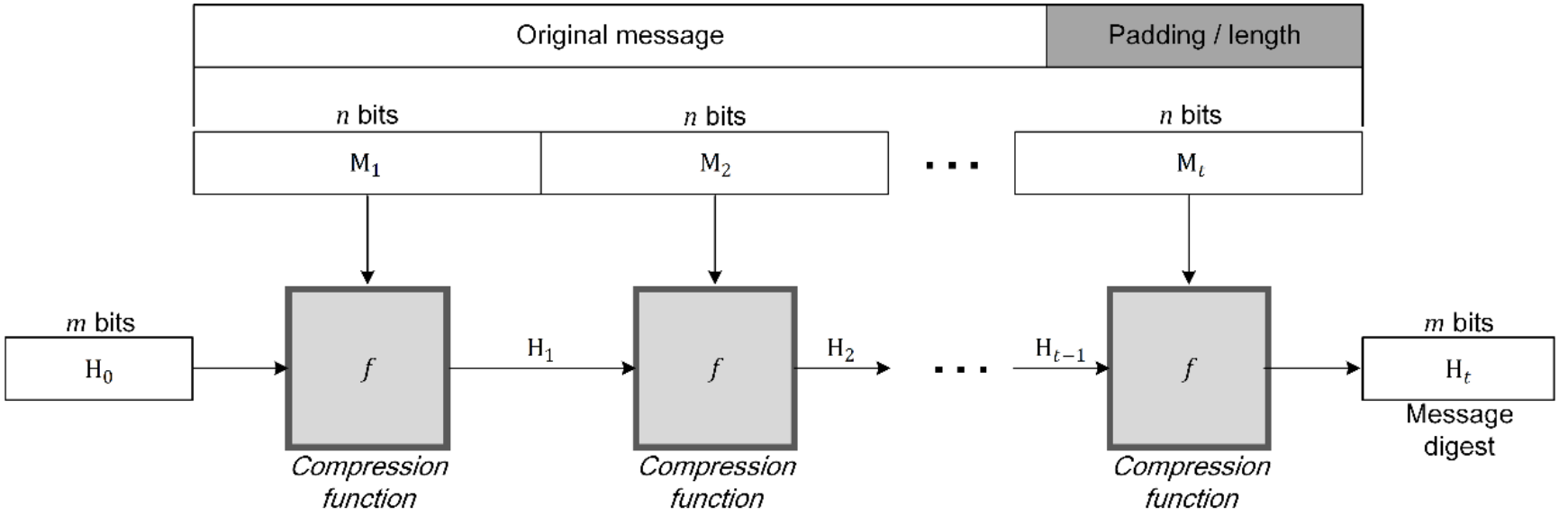
*역상(Preimage)
어떤 출력값을 만들어내는 원래 입력값

개요

- 압축 함수(Compression Function)
  - 고정 길이의 입력을 받아 더 짧고 고정된 길이 출력으로 변환하는 함수
- 반복 암호학적 해시함수(Iterated Cryptographic Hash Function) (1/4)
  - 압축 함수를 반복적으로 적용하여 고정된 크기의 해시 값을 생성하는 해시함수

개요

- 반복 암호학적 해시함수 (2/4)
  - Merkle-Damgård 구조 (Merkle-Damgård scheme) (1/2)
    - 충돌 저항성을 가지는 압축 함수로부터 충돌 저항성을 갖는 해시 함수를 구축하는 구성 방식



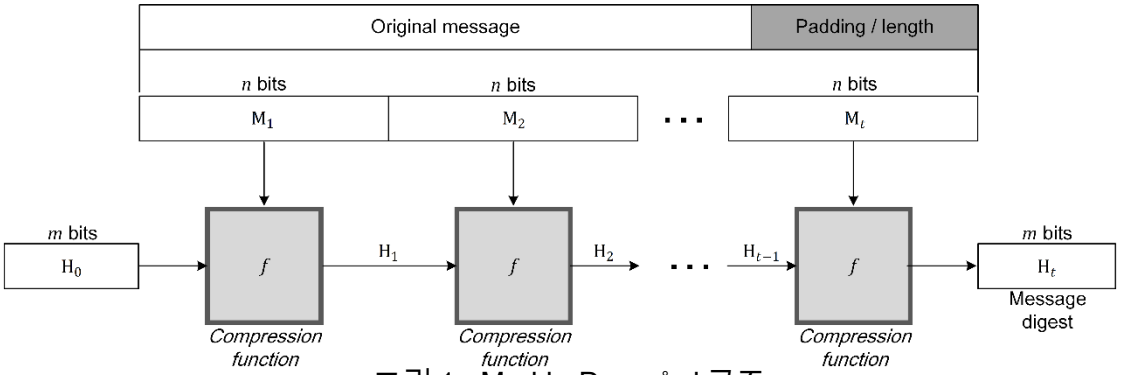
<그림 1> Merkle-Damgård 구조

개요

- 반복 암호학적 해시함수 (3/4)

*다이제스트(Digest)
해시함수의 출력 값 (해시 값)

  - Merkle-Damgård 구조 (Merkle-Damgård scheme) (2/2)
    - 과정
      1. 메시지에 패딩을 붙여 n비트를 갖는 t개의 블록으로 구성
         - 각 블록은 $M_1, M_2, \dots, M_t$로 정의
         - t번 반복하는 동안에 생성된 다이제스트를 $H_0, H_1, \dots, H_t$로 정의
      2. 반복을 시작하기 전, H_0는 고정된 값을 갖도록 설정
         - 보통 H_0를 초기 벡터 IV라고 정의
      3. 매 반복과정에서 압축 함수는 H_{i-1}과 M_i를 이용해 H_i를 생성
         → $H_i = f(H_{i-1}, M_i)$ (f는 압축 함수)
      4. 최종적으로 생성된 H_t가 원래 메시지로부터 생성된 암호학적 해시함수의 해시 값
         → $H_t = h(M)$



<그림 1> Merkle-Damgård 구조

개요

- 반복 암호학적 해시함수 (4/4)
  - 분류
    - 전용 설계 압축 함수 기반 해시함수
      - 해시 목적에 맞게 압축 함수를 별도로 설계
      - e.g., 메시지 다이제스트, SHA, RIPEMD, HAVAL
    - 블록 암호 기반 해시함수
      - 압축 함수로 대칭-키 블록 암호 사용
      - e.g., Rabin 구조, Davies-Meyer 구조, Matyas-Meyer-Oseas 구조, Miyaguchi-Preneel 구조

*메시지 다이제스트(MD, Message Digest)
*SHA (Secure Hash Algorithm)
*RIPEMD (RACE Integrity Primitives Evaluation Message Digest)
*HAVAL (Hash of Variable Length)

개요

- 반복 암호학적 해시함수 – 분류
  - 전용 설계 압축 함수 기반 해시함수 (1/6)
    - 메시지 다이제스트
      - 개요
        - Ron Rivest가 설계한 해시함수 알고리즘 계열
      - 특징
        - 128비트 메시지 다이제스트 생성
        - 128비트 길이의 충돌 저항성 취약
      - 대표 알고리즘
        - MD2, MD4, MD5

[표 1] 메시지 다이제스트 특성

특징	MD2	MD4	MD5
Max Message size	임의 길이	2^{64}-1 bit	2^{64}-1 bit
Block size	128 bit	512 bit	512 bit
Message digest size	128 bit	128 bit	128 bit
Number of rounds	18 rounds	3 rounds / 48 steps	4 rounds / 64 steps
Word size	8	32	32

개요

- 반복 암호학적 해시함수 – 분류
  - 전용 설계 압축 함수 기반 해시함수 (2/6)
    - SHA (1/3)
      - 개요

*FIPS (Federal Information Processing Standard)

        - NIST (National Institute of Standards and Technology)에서 개발한 표준 (FIPS 180)
        - SHS (Secure Hash Standard)으로도 지칭
      - 특징
        - 초기 SHA 계열은 MD 계열 해시함수와 유사한 반복 구조를 사용
          - 메시지 확장을 통해 워드간 XOR 및 회전으로 상호 연결되어 확산 강화
        - 보안성 및 출력 구조에 따라 SHA-1, SHA-2, SHA-3로 발전
          - SHA-3에서는 스펀지 구조로 분화

개요

- 반복 암호학적 해시함수 – 분류
  - 전용 설계 압축 함수 기반 해시함수 (3/6)
    - SHA (2/3)
      - 대표 알고리즘 (1/2)
        - SHA-1
          - 1995년 개정된 FIPS 180-1에 포함
          - 2005년 충돌 공격 가능성 발견
        - SHA-2 (SHA-224, SHA-256, SHA-384, SHA-512)
          - FIPS 180-2로 개정되며 4개의 버전 포함

[표 2] 안전 해시 알고리즘(SHA)의 특성

특징	SHA-1	SHA-224	SHA-256	SHA-384	SHA-512
Max Message size	2^{64}-1 bit	2^{64}-1 bit	2^{64}-1 bit	2^{128}-1 bit	2^{128}-1 bit
Block size	512	512	512	1024	1024
Message digest size	160	224	256	384	512
Number of rounds	80	64	64	80	80
Word size	32	32	32	64	64

개요

- 반복 암호학적 해시함수 – 분류

*XOF (Extendable-output function)

  - 전용 설계 압축 함수 기반 해시함수 (4/6)
    - SHA (3/3)
      - 대표 알고리즘 (2/2)
        - SHA-3 (Keccak)
          - SHA-1의 충돌저항성에 대한 우려로 새롭게 선정된 표준 (FIPS 202)
            - 고정 길이 해시함수: SHA3-224, SHA3-256, SHA3-384, SHA3-512
            - 가변 길이 해시함수(XOF): SHAKE128, SHAKE256
          - 해시 값으로 임의 길이의 비트열 생성 가능
          - 이전 SHA와는 다른 스펀지 구조 사용
            - 패딩 후 흡수(Absorbing)와 추출(Squeezing) 단계로 해시 값 출력

[표 3] SHA3의 특성

특징	SHA3-224	SHA3-256	SHA3-384	SHA3-512
Max Message size	2^{64}-1 bit	2^{64}-1 bit	2^{128}-1 bit	2^{128}-1 bit
Block size	1152	1088	832	576
Message digest size	224	256	384	512
Number of rounds	24	24	24	24
Word size	64	64	64	64

개요

- 반복 암호학적 해시함수 – 분류
  - 전용 설계 압축 함수 기반 해시함수 (5/6)
    - RIPEMD
      - 개요
        - MD 계열 해시함수의 보안성을 보완하기 위해 설계된 해시 알고리즘 계열
      - 특징
        - 두 개의 독립적인 병렬 처리 라인에서 서로 다른 순서로 메시지를 처리한 뒤 결합하는 구조
      - 대표 알고리즘
        - RIPEMD-128, RIPEMD-160, RIPEMD-256, RIPEMD-320

[표 4] RIPEMD의 특성

특징	RIPEMD-128	RIPEMD-160	RIPEMD-256	RIPEMD-320
Max Message size	2^{64}-1 bit	2^{64}-1 bit	2^{64}-1 bit	2^{64}-1 bit
Block size	512 bit	512 bit	512 bit	512 bit
Message digest size	128 bit	160 bit	256 bit	320 bit
Number of rounds	4 rounds / 64 steps	5 rounds / 80 steps	4 rounds / 64 steps	5 rounds / 80 steps
Word size	32 bit	32 bit	32 bit	32 bit

개요

- 반복 암호학적 해시함수 – 분류
  - 전용 설계 압축 함수 기반 해시함수(6/6)
    - HAVAL
      - 개요
        - 출력 길이와 라운드 수를 선택할 수 있도록 설계된 해시 알고리즘
      - 특징
        - 1024비트 메시지 블록 단위 처리
        - 128, 160, 192, 224, 256 비트 메시지 다이제스트 생성
        - 3, 4, 5회 중 처리 단계 선택 가능
      - 대표 알고리즘
        - HAVAL-128, HAVAL-160, HAVAL-192, HAVAL-224, HAVAL-256

[표 5] HAVAL의 특성

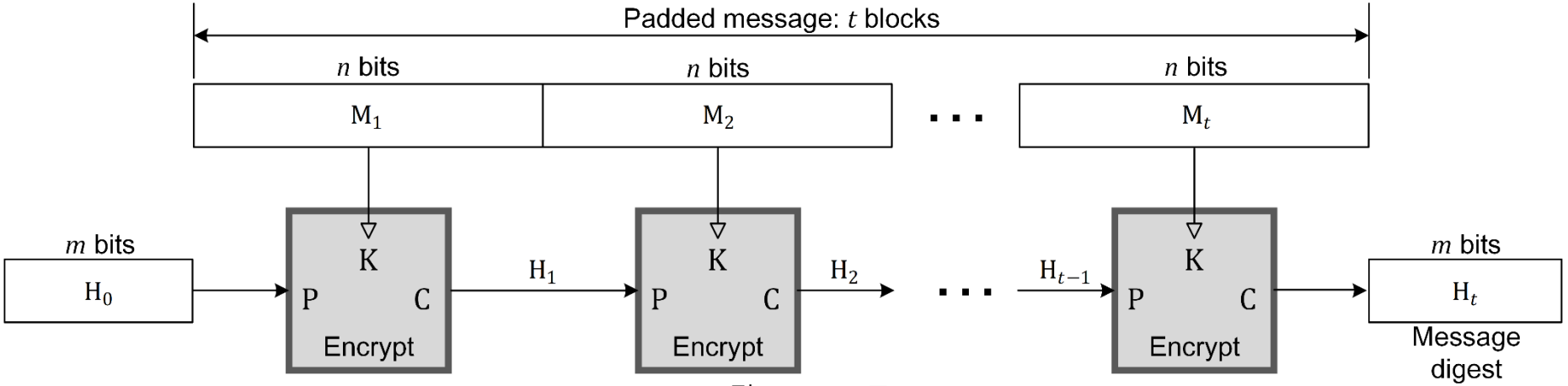
특징	HAVAL-128	HAVAL-160	HAVAL-192	HAVAL-224	HAVAL-256
Max Message size	2^{64}-1 bit	2^{64}-1 bit	2^{64}-1 bit	2^{64}-1 bit	2^{64}-1 bit
Block size	1024 bit	1024 bit	1024 bit	1024 bit	1024 bit
Message digest size	128 bit	160 bit	192 bit	224 bit	256 bit
Number of rounds	3/4/5 passes	3/4/5 passes	3/4/5 passes	3/4/5 passes	3/4/5 passes
Word size	32 bit	32 bit	32 bit	32 bit	32 bit

개요

- 반복 암호학적 해시함수 – 분류

*중간 일치 공격(MITM, Meet-In-the-Middle Attack)
목표값을 향해 정방향과 역방향으로 동시에 계산해 중간에서 일치하는 값을 찾는 공격

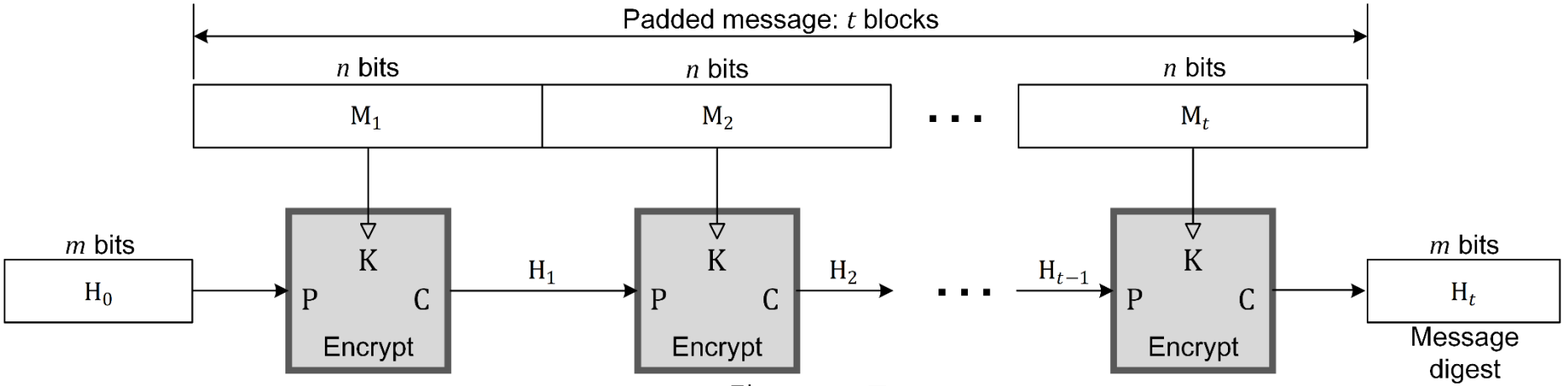
  - 블록 암호 기반 해시함수 (1/8)
    - Rabin 구조(Rabin Scheme) (1/2)
      - 개요
        - Merkle-Damgård 구조를 기반으로 한 블록 암호 기반 해시 구조
        - 블록 암호의 암호화 기능을 압축 함수로 사용
      - 특징
        - 암호화 결과를 변형 없이 다음 다이제스트로 사용
        - 다이제스트 크기 = 사용되는 블록 암호의 블록 크기
        - 복호화 알고리즘 사용 가능성으로 인한 중간 일치 공격 취약성



<그림 2> Rabin 구조

개요

- 반복 암호학적 해시함수 – 분류
  - 블록 암호 기반 해시함수 (2/8)
    - Rabin 구조(Rabin Scheme) (2/2)
      - 동작 방식
        - 메시지 블록 M_i를 암호화 키 K로 사용
        - 이전 다이제스트 H_{i-1}를 평문 P로 사용
        - 암호문 C를 새로운 다이제스트 H_i로 사용
      - 수식
        - $H_i = E_{M_i}(H_{i-1})$



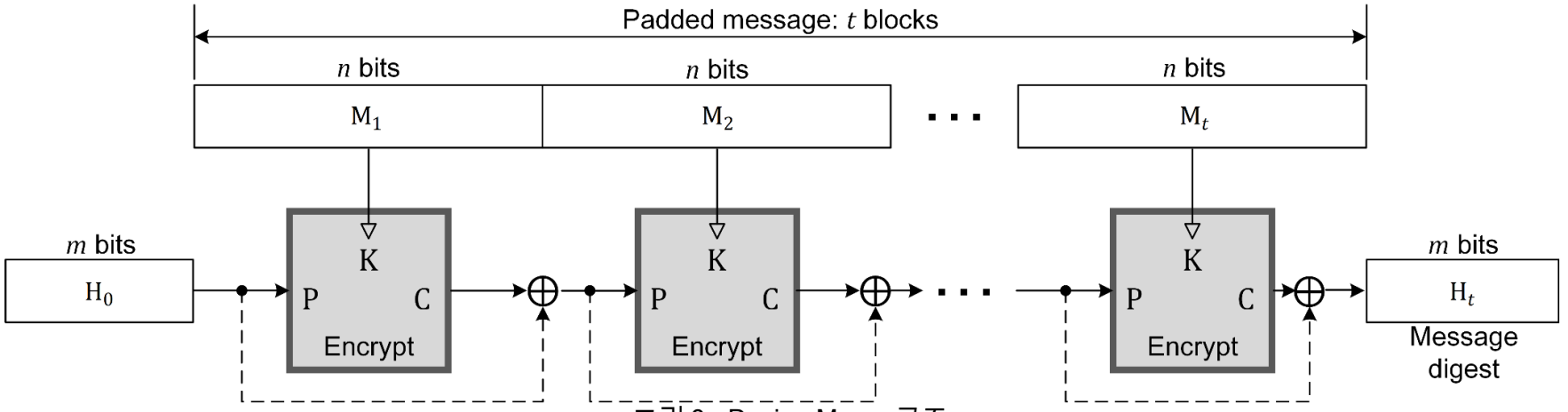
<그림 2> Rabin 구조

개요

- 반복 암호학적 해시함수 – 분류

*포워드 피드(Forward Feed)
암호화 결과만 쓰지 않고, 입력으로 들어간 이전 다이제스트를 다시 XOR하는것

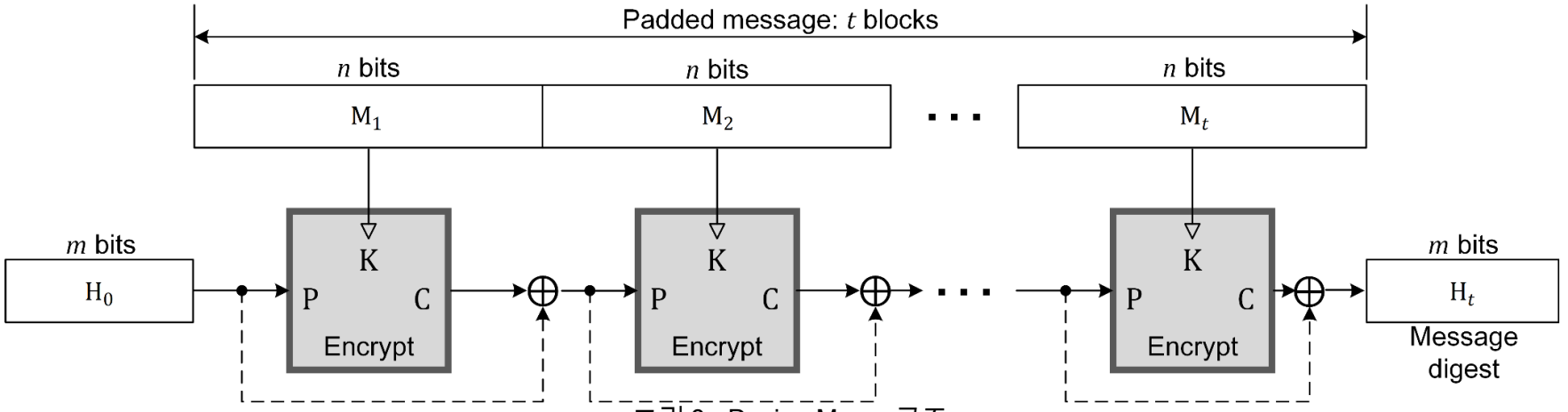
  - 블록 암호 기반 해시함수 (3/8)
    - Davies-Meyer 구조(Davies-Meyer Scheme) (1/2)
      - 개요
        - Rabin 구조를 개선한 블록 암호 기반 해시 구조
        - 중간 일치 공격을 완화하기 위해 포워드 피드 기능 사용
      - 특징
        - 포워드 피드를 통한 역연산 차단
        - 이상적 블록 암호의 가정 하에 역상·충돌 저항성이 증명된 구조



<그림 3> Davies-Meyer 구조

개요

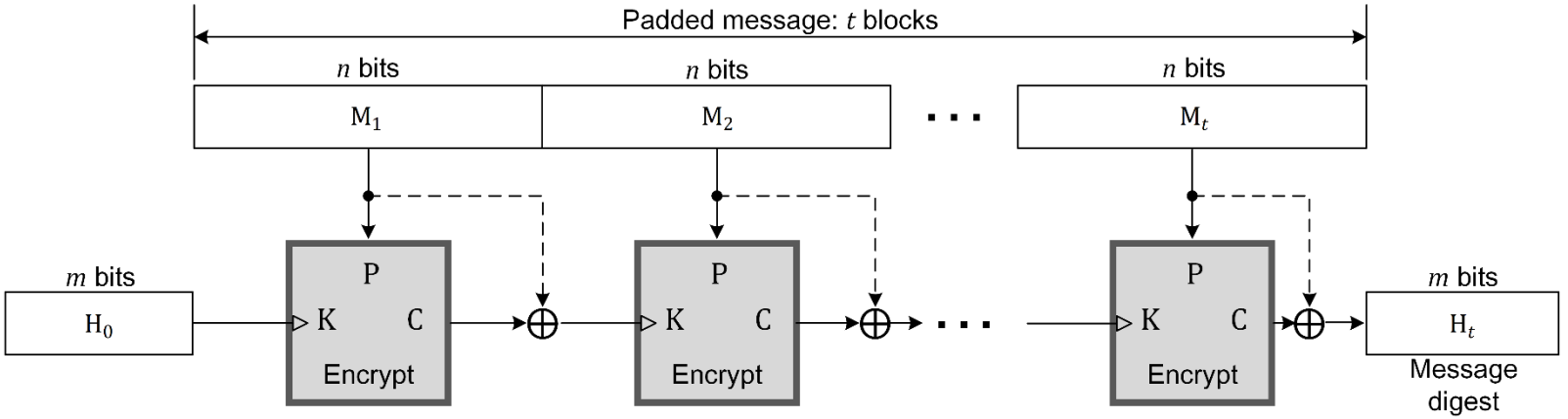
- 반복 암호학적 해시함수 – 분류
  - 블록 암호 기반 해시함수 (4/8)
    - Davies-Meyer 구조(Davies-Meyer Scheme) (2/2)
      - 동작 방식
        - 메시지 블록 M_i를 암호화 키 K로 사용
        - 이전 다이제스트 H_{i-1}를 평문 P로 사용
        - 암호화 결과와 이전 다이제스트를 XOR하여 새로운 다이제스트 생성
      - 수식
        - $H_i = E_{M_i}(H_{i-1}) \oplus H_{i-1}$



<그림 3> Davies-Meyer 구조

개요

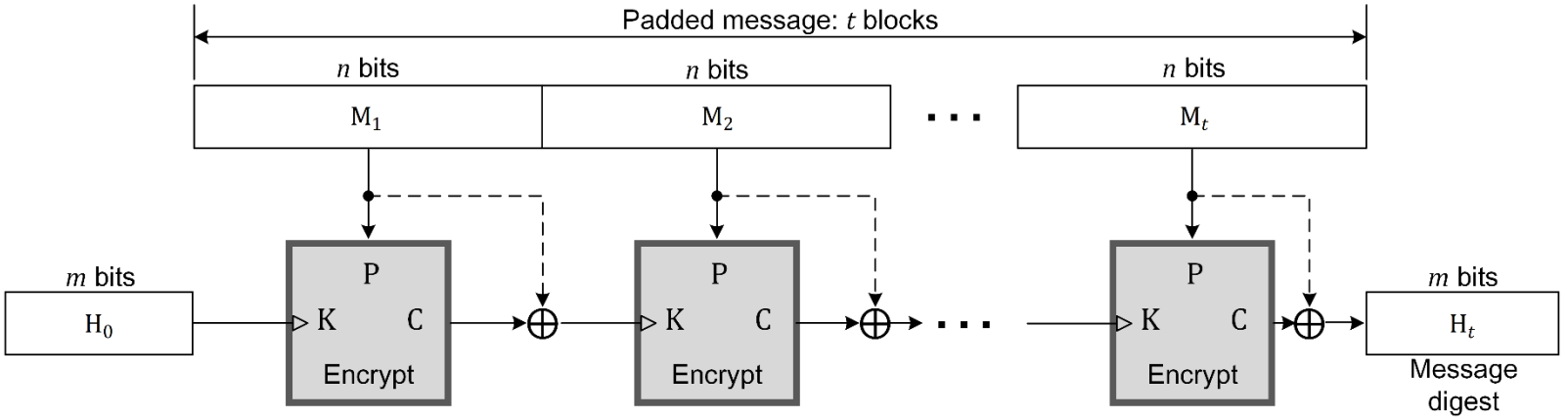
- 반복 암호학적 해시함수 – 분류
  - 블록 암호 기반 해시함수 (5/8)
    - Matyas-Meyer-Oseas 구조(Matyas-Meyer-Oseas Scheme) (1/2)
      - 개요
        - Davies-Meyer 구조의 변형
        - 메시지 블록과 이전 다이제스트의 역할을 바꾼 구조
      - 특징
        - 메시지 블록과 암호 키의 길이가 같을 때 사용 가능
          - 길이가 다를 경우 키 크기에 맞춰 변환 함수 g 수행
        - 메시지 블록을 포워드 피드 값으로 사용



<그림 4> Matyas-Meyer-Oseas 구조

개요

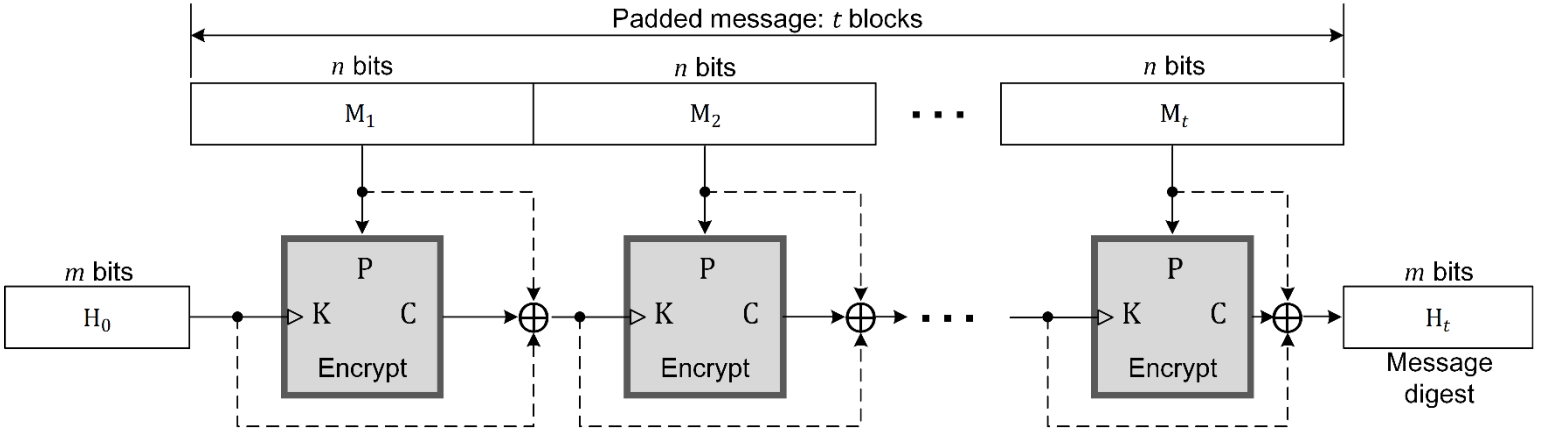
- 반복 암호학적 해시함수 – 분류
  - 블록 암호 기반 해시함수 (6/8)
    - Matyas-Meyer-Oseas 구조(Matyas-Meyer-Oseas Scheme) (2/2)
      - 동작 방식
        - 이전 다이제스트 H_{i-1}를 암호화 키 K로 사용
        - 메시지 블록 M_i를 평문 P로 사용
        - 암호화 결과와 메시지 블록을 XOR하여 새로운 다이제스트 생성
      - 수식
        - $H_i = E_{H_{i-1}}(M_i) \oplus M_i$
        - $H_i = E_{g(H_{i-1})}(M_i) \oplus M_i$ (크기 변환이 필요한 경우)



<그림 4> Matyas-Meyer-Oseas 구조

개요

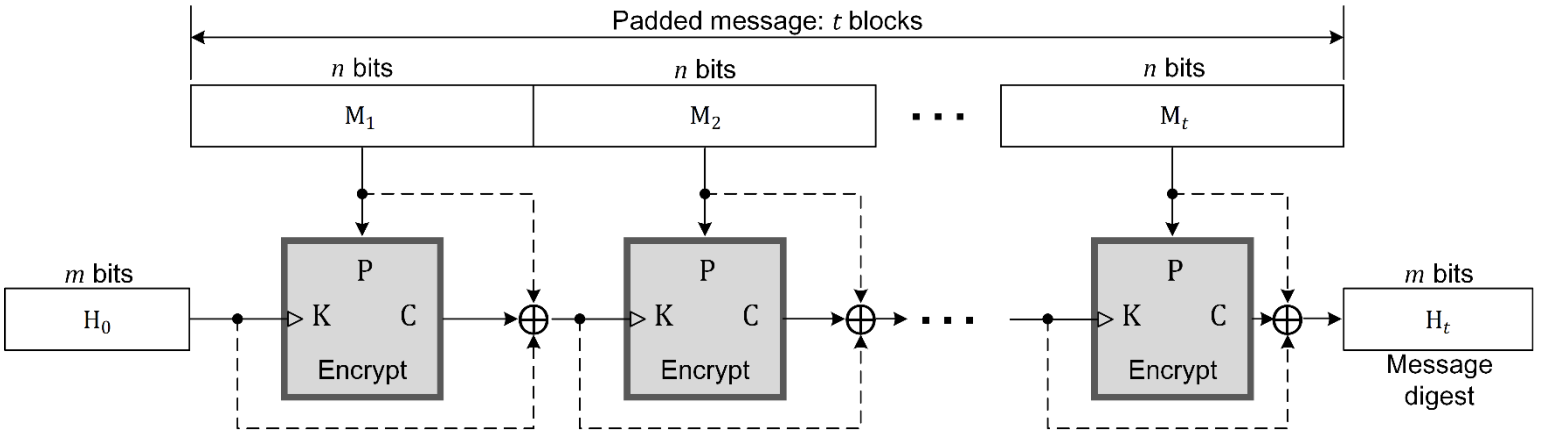
- 반복 암호학적 해시함수 – 분류
  - 블록 암호 기반 해시함수 (7/8)
    - Miyaguchi-Preneel 구조(Miyaguchi-Preneel Scheme) (1/2)
      - 개요
        - Matyas-Meyer-Oseas 구조를 확장한 블록 암호 기반 해시 구조
        - 평문, 암호 키, 암호문을 모두 XOR하여 새로운 다이제스트 생성
      - 특징
        - Matyas-Meyer-Oseas 구조보다 더 많은 값을 결합
        - 암호문, 평문, 키 값을 모두 반영한 다이제스트 생성
        - WHIRLPOOL 해시함수에서 사용되는 구조



<그림 5> Miyaguchi-Preneel 구조

개요

- 반복 암호학적 해시함수 – 분류
  - 블록 암호 기반 해시함수 (8/8)
    - Miyaguchi-Preneel 구조(Miyaguchi-Preneel Scheme) (2/2)
      - 동작 방식
        - 이전 다이제스트 H_{i-1}를 암호화 키 K로 사용
        - 메시지 블록 M_i를 평문 P로 사용
        - 암호화 결과, 메시지 블록, 이전 다이제스트를 모두 XOR
      - 수식
        - $H_i = E_{H_{i-1}}(M_i) \oplus M_i \oplus H_{i-1}$



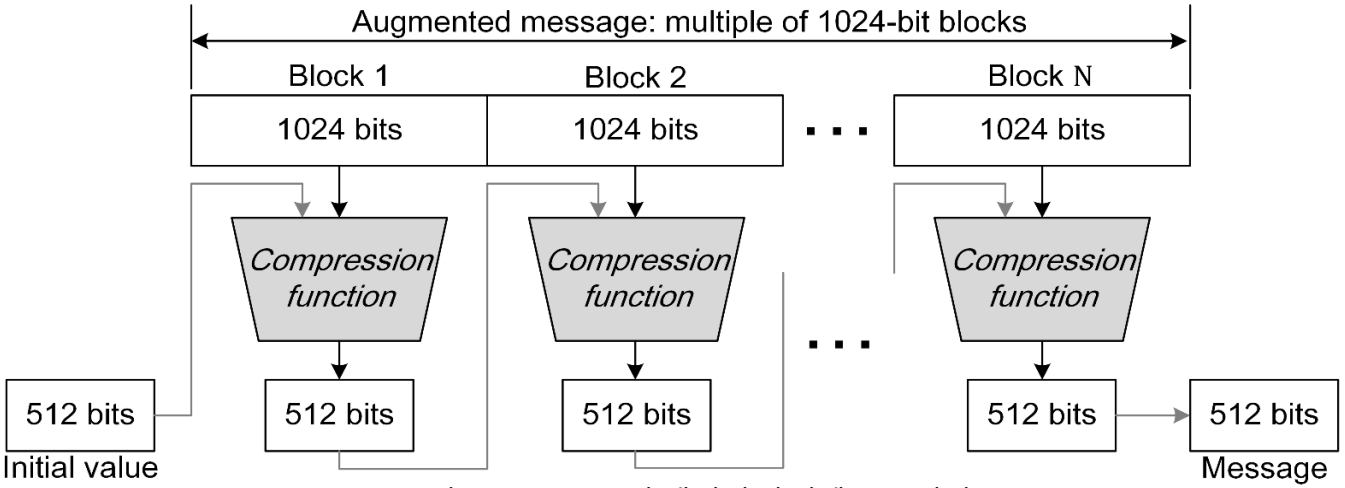
<그림 5> Miyaguchi-Preneel 구조

목 차

- 개요
- SHA-512
- WHIRLPOOL

SHA-512

- 개요
  - 512비트 메시지 다이제스트를 생성하는 SHA 계열 해시 알고리즘
  - Merkle-Damgård 구조 기반
  - 1024비트 메시지 블록 단위 처리
  - 512비트 초기값을 사전에 지정하여 사용



<그림 6> SHA-512의 메시지 다이제스트 생성

SHA-512

- 동작 과정

1. 입력 메시지 전처리
   - 전체 메시지를 1024비트 메시지 블록으로 확장 및 분할
   - 초기값 H_0 설정
2. 압축 함수 연산
   a. 메시지 워드 생성
      - 메시지 블록을 80개의 64비트 워드로 확장
   b. 압축 함수를 통해 80 라운드 연산 수행
   c. 최종 가산(Final Adding) 수행
      - 80 라운드로 생성된 값에 이전 중간 다이제스트 값을 덧셈 연산
   d. 중간 다이제스트 갱신
      - 최종 가산 결과를 새로운 512비트 중간 다이제스트로 사용
3. 모든 메시지 블록에 대해 2번 과정을 반복

SHA-512

- 입력 메시지 전처리 (1/5)
  - 메시지 준비
    - 메시지의 길이가 2^{128} 비트 미만인지 확인
      - 메시지의 길이가 2^{128} 비트 이상인 경우 처리 불가능
        - 2^{128} 비트는 대부분의 컴퓨터 시스템의 저장 용량 이상의 크기
    - 예제 12.1

      길이가 2^{128}비트인 메시지를 전송하려고 할 때 통신네트워크 시스템에서 초당 2^{64}비트를 전송할 수 있다면 이 메시지를 전송하는 데 얼마나 걸리는가?

      - 전송 시간 = 전체 메시지 크기(bit) / 초당 전송량 (bit/sec)
        - $2^{128} \div 2^{64} = 2^{64}$
      - 전송에 걸리는 시간: 2^{64} 초 ≈약 5849억 년
    - 예제 12.2

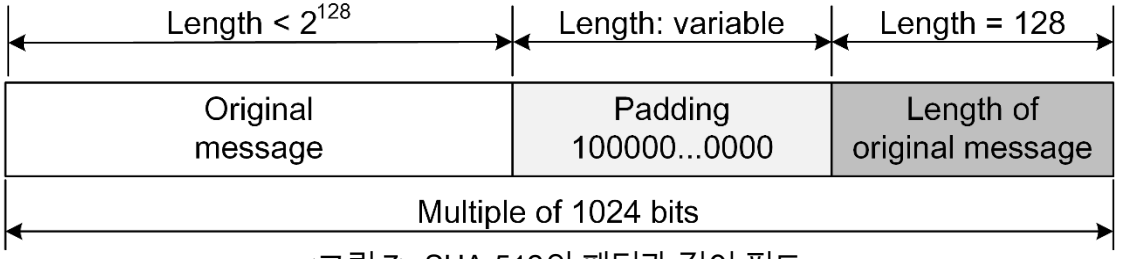
      길이가 2^{128}비트인 메시지는 몇 페이지인가?

      - 한 문자가 64비트 이고, 한 페이지의 크기를 2^{18} 비트라고 가정
      - 페이지 수 = 전체 메시지 크기 / 한 페이지 크기
        - $2^{128} \div 2^{18} = 2^{110}$
      - $2^{110} \approx 1.3 \times 10^{33}$의 큰 수

SHA-512

- 입력 메시지 전처리 (2/5)
  - 길이 필드와 패딩 (1/3)
    - 길이 필드
      - 128비트의 부호 없는 정수 길이 필드 추가
        - 메시지 최대 길이인 0 ~ 2^{128}-1 내에서 표현
      - 패딩과 길이 필드를 제외한 원래 메시지 길이 정보 삽입
    - 패딩
      - 길이 필드와 패딩을 추가한 메시지의 길이가 1024비트의 배수가 되도록 패딩
      - 첫 번째 비트는 1, 이어지는 비트는 모두 0
      - 패딩 필드 길이 계산
        $(|M| + |P| + 128) = 0 \; mod \; 1024 \rightarrow |P| = (-|M| - 128) \; mod \; 1024$

$|M|$: 원래 메시지 길이
$|P|$: 패딩 필드 길이



<그림 7> SHA-512의 패딩과 길이 필드

SHA-512

- 입력 메시지 전처리 (3/5)
  - 길이 필드와 패딩 (2/3)
    - 예제 12.3

      원래 메시지의 길이가 2590비트라면 추가되어야 할 패딩 비트는 얼마인가?

      - $|P| = (-2590 - 128)\ mod\ 1024 = -2718\ mod\ 1024 = 354$

      ∴ 패딩 길이는 354비트이며, 1개의 1과 353개의 0으로 이루어짐
    - 예제 12.4

      원래 메시지의 길이가 이미 1024의 배수가 될 경우에도 패딩이 필요한가?

      - 필요
        - 길이 필드를 추가해야하므로 길이가 1024비트인 새로운 블록이 필요함

SHA-512

- 입력 메시지 전처리 (4/5)
  - 길이 필드와 패딩 (3/3)
    - 예제 12.5

메시지에 추가되는 패딩 비트의 최소 크기와 최대 크기는 얼마나 되는가?

- 최소 크기: 0비트
  - $(-M - 128)\ mod\ 1024\ =\ 0$ 인 경우 가능
    - $|M| \equiv -128\ mod\ 1024 \equiv 896\ mod\ 1024$
    - 마지막 블록이 896비트인 경우, 마지막 블록이 1024 비트가 되도록 128비트의 길이 비트를 추가하면 896+128=1024 이므로 패딩이 불필요

- 최대 크기: 1023비트
  - $(-M - 128) = 1023\ mod\ 1024$ 인 경우 가능
    - $|M| \equiv 897\ mod\ 1024$
    - 마지막 블록이 897비트인 경우, 128비트 길이 필드를 바로 추가하면 897 + 128 = 1025 비트로 1비트가 초과되어 추가로 1023비트 패딩 필요

SHA-512

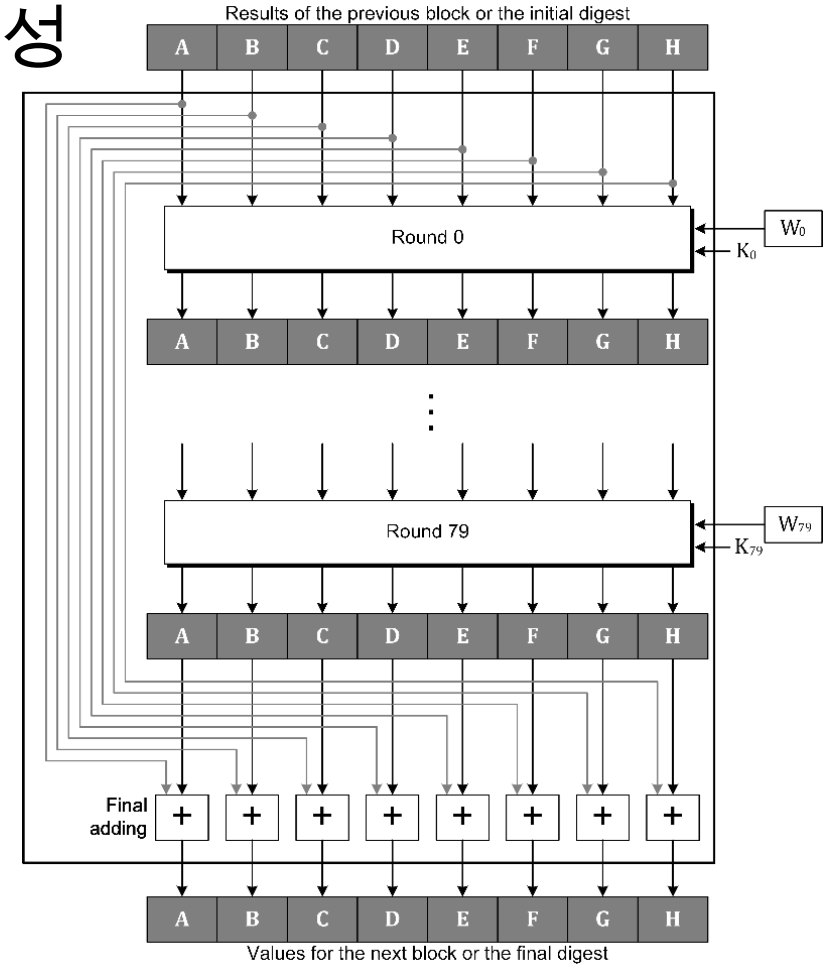
- 입력 메시지 전처리 (5/5)
  - 초기값 H_0 설정
    - SHA-512 압축 과정을 시작하기 위한 초기값
      - 소수 2, 3, 5, 7, 11, 13, 17, 19로 만들어진 수
      - 해당 소수의 제곱근을 구한 다음 소수점 이하만 취득
    - 항상 고정되어있는 8개의 64비트 상수를 사용
      - A_0~H_0의 값은 표준에 의해 고정
    - 첫 번째 메시지 블록 처리 시 초기 중간 다이제스트로 사용
      - 이후 압축 과정에서 메시지에 따라 A~H값 갱신

[표 6] SHA-512에서 메시지 다이제스트 초기화 상수값

Buffer	Value (in hexadecimal)	Buffer	Value (in hexadecimal)
A_0	6A09E667F3BCC908	E_0	510E527FADE682D1
B_0	BB67AE8584CAA73B	F_0	9B05688C2B3E6C1F
C_0	3C6EF372EF94F82B	G_0	1F83D9ABFB41BD6B
D_0	A54FE53A5F1D36F1	H_0	5BE0CD19137E2179

SHA-512

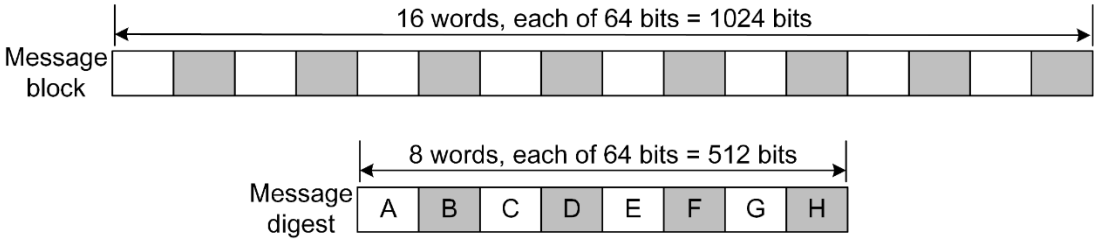
- 압축 함수 연산 (1/9)
  - 모든 블록에 대해 반복 연산
    1. 각 메시지 블록에 대한 워드 생성
    2. 80 라운드 연산
    3. 최종 가산 수행
    4. 중간 다이제스트 갱신



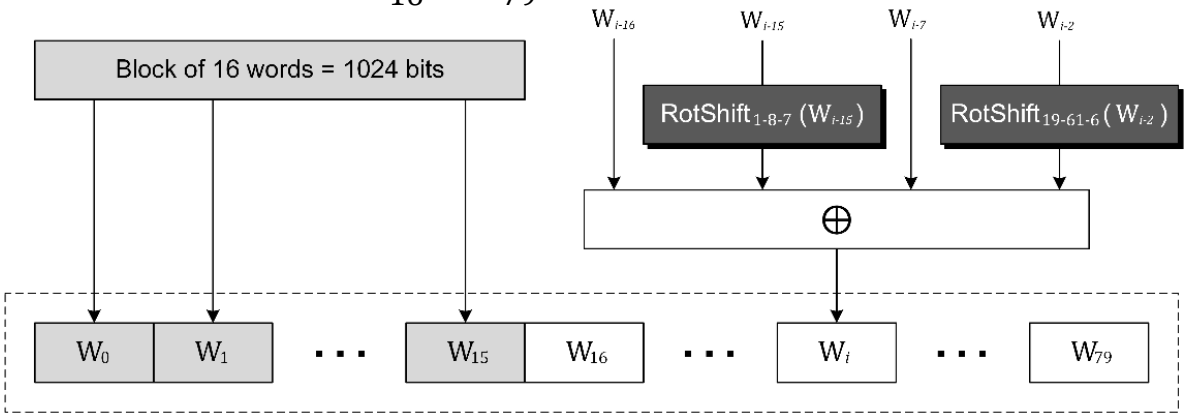
<그림 8> SHA-512의 압축 함수

SHA-512

- 압축 함수 연산 (2/9)
  - 메시지 워드 생성 (1/2)
    - 워드
      - SHA-512는 워드 단위로 계산
      - 1개 워드를 64비트로 정의
      - 메시지 다이제스트는 워드 8개로 구성
    - 워드 확장
      - 1024비트 메시지 블록을 80개의 64비트 워드로 확장
        - W_0~W_{15} : 1024 비트 메시지 블록을 64비트씩 나누어 초기 워드 16개 생성
        - W_{16}~W_{79}: 이전에 생성된 워드들을 회전, 시프트, XOR, 덧셈 연산을 적용하여 생성



<그림 9> SHA-512의 워드로서의 메시지 블록과 다이제스트



Rotshift$_{l-m-n}$: $\boldsymbol{RotR_l(x) \oplus RotR_m(x) \oplus ShR_n(x)}$

$\boldsymbol{RotR_i(x)}$: Right-rotation of the argument x by i bits

$\boldsymbol{ShR_i(x)}$ Shift-right of the argument x by i bits and padding the left by 0's

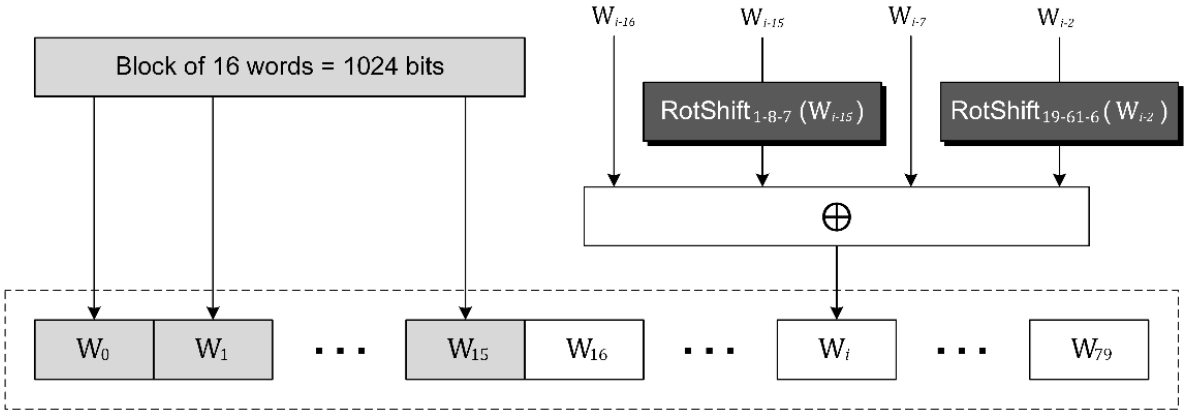
<그림 10> SHA-512의 워드확장

SHA-512

- 압축 함수 연산 (3/9)
  - 메시지 워드 생성 (2/2)
    - 예제 12.6

워드 W_{60}이 어떻게 만들어지는지 보이시오

- 각 워드 W_{16}부터 W_{79}는 사전에 만들어진 4개의 워드로부터 만들어짐
  $\therefore W_{60} = W_{44} \oplus \text{RotShift}_{1-8-7}(W_{45}) \oplus W_{53} \oplus \text{RotShift}_{19-61-6}(W_{58})$



Rotshift$_{1-m-n}$: $\boldsymbol{RotR_l(x) \oplus RotR_m(x) \oplus ShR_n(x)}$
$\boldsymbol{RotR_i(x)}$: Right-rotation of the argument x by i bits
$\boldsymbol{ShR_i(x)}$ Shift-right of the argument x by i bits and padding the left by 0's

<그림 10> SHA-512의 워드확장

SHA-512

- 압축 함수 연산 (4/6)
  - 80 라운드 연산 (1/4)
    - 라운드 구성요소
      - 주 함수(Majority Function)
        - 3개의 버퍼(A, B, C)에서 대응되는 3개의 비트에 대해 다수결 연산 수행
          - $(x \text{ AND } y) \oplus (y \text{ AND } z) \oplus (z \text{ AND } x)$
      - 조건부 함수(Conditional Function)
        - 3개의 버퍼(E, F, G) 중 E의 비트 값에 따라 F 또는 G 선택
          - $(x \text{ AND } y) \oplus (\text{NOT } x \text{ AND } z)$
      - 순환 함수(Rotate Function)
        - 동일한 버퍼의 3개의 값을 오른쪽으로 순환 후 XOR 연산 적용
          - $Rotate\ (A)$: $RotR_{28}(A) \oplus RotR_{34}(A) \oplus RotR_{39}(A)$
          - $Rotate\ (E)$: $RotR_{14}(E) \oplus RotR_{18}(E) \oplus RotR_{41}(E)$
      - 오른쪽 순환 함수(Right-Rotation Function)
        - 입력 값의 i개 비트를 오른쪽으로 순환
          - $\text{RotR}_i(x)$

SHA-512

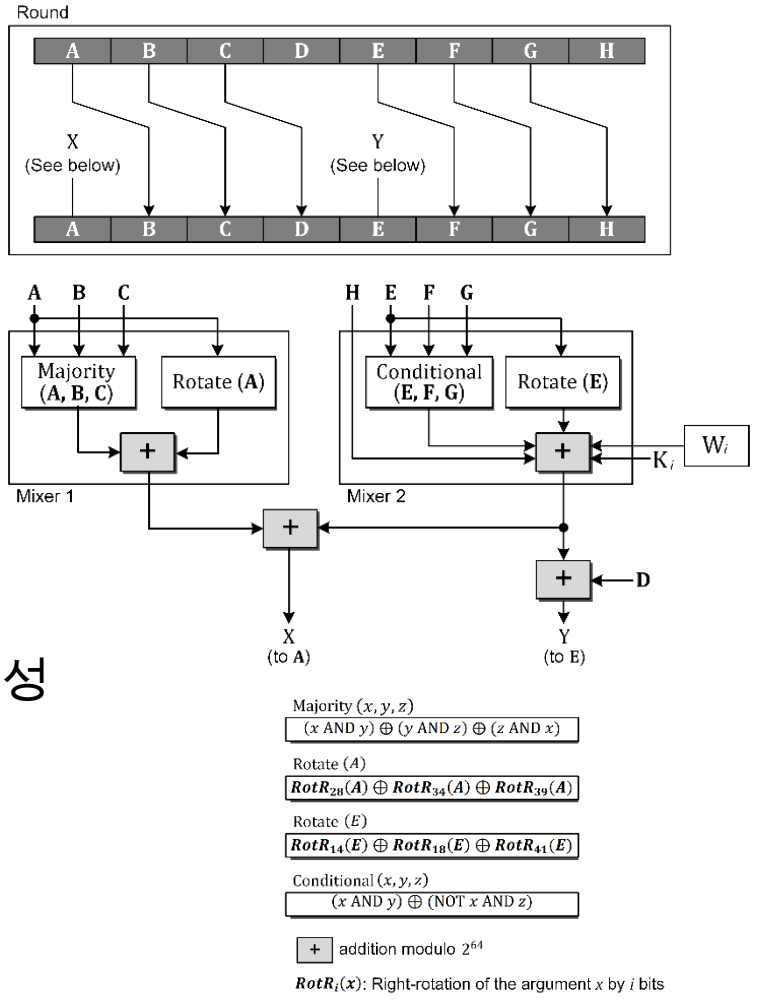
- 압축 함수 연산 (5/9)
  - 80 라운드 연산 (2/4)
    - 라운드 구성요소
      - 합 연산
        - 모듈로 2^{64}을 합으로 하는 절차에 사용
        - 2개 이상의 버퍼를 합한 결과가 항상 64비트 워드
      - 라운드 상수 K_t
        - K_0부터 K_{79}까지 80개의 64비트 상수로 구성
        - 각 라운드마다 하나의 상수 K_t 사용
        - 80개 소수의 세제곱근에서 소수점 이하 64비트를 취해 생성

[표 7] SHA-512의 80 라운드에서 사용하는 80개의 상수

428A2F98D728AE22	7137449123EF65CD	B5C0FBCFEC4D3B2F	E9B5DBA58189DBBC
3956C25BF348B538	59F111F1B605D019	923F82A4AF194F9B	AB1C5ED5DA6D8118
D807AA98A3030242	12835B0145706FBE	243185BE4EE4B28C	550C7DC3D5FFB4E2
72BE5D74F27B896F	80DEB1FE3B1696B1	9BDC06A725C71235	C19BF174CF692694
E49B69C19EF14AD2	EFBE4786384F25E3	0FC19DC68B8CD5B5	240CA1CC77AC9C65
2DE92C6F592B0275	4A7484AA6EA6E483	5CB0A9DCBD41FBD4	76F988DA831153B5
983E5152EE66DFAB	A831C66D2DB43210	B00327C898FB213F	BF597FC7BEEF0EE4
C6E00BF33DA88FC2	D5A79147930AA725	06CA6351E003826F	142929670A0E6E70
27B70A8546D22FFC	2E1B21385C26C926	4D2C6DFC5AC42AED	53380D139D95B3DF
650A73548BAF63DE	766A0ABB3C77B2A8	81C2C92E47EDAEE6	92722C851482353B
A2BFE8A14CF10364	A81A664BBC423001	C24B8B70D0F89791	C76C51A30654BE30
D192E819D6EF5218	D69906245565A910	F40E35855771202A	106AA07032BBD1B8
19A4C116B8D2D0C8	1E376C085141AB53	2748774CDF8EEB99	34B0BCB5E19B48A8
391C0CB3C5C95A63	4ED8AA4AE3418ACB	5B9CCA4F7763E373	682E6FF3D6B2B8A3
748F82EE5DEFB2FC	78A5636F43172F60	84C87814A1F0AB72	8CC702081A6439EC
90BEFFFA23631E28	A4506CEBDE82BDE9	BEF9A3F7B2C67915	C67178F2E372532B
CA273ECEEA26619C	D186B8C721C0C207	EADA7DD6CDE0EB1E	F57D4F7FEE6ED178
06F067AA72176FBA	0A637DC5A2C898A6	113F9804BEF90DAE	1B710B35131C471B
28DB77F523047D84	32CAAB7B40C72493	3C9EBE0A15C9BEBC	431D67C49C100D4C
4CC5D4BECB3E42B6	4597F299CFC657E2	5FCB6FAB3AD6FAEC	6C44198C4A475817

SHA-512

- 압축 함수 연산 (6/9)
  - 80 라운드 연산 (3/4)
    - 라운드 구조 (1/2)
      - 입력값 설정
        - 8개의 64비트 버퍼 A~H를 입력으로 사용
        - B, C, D, F, G, H는 이전 버퍼 값 이동으로 생성
          - $A \rightarrow B, B \rightarrow C, C \rightarrow D$
          - $E \rightarrow F, F \rightarrow G, G \rightarrow H$
        - A(=X)와 E(=Y)는 혼합 연산을 통해 새로 생성
          - A: Mixer 1과 Mixer 2의 결과를 더해 생성
          - E: D와 Mixer 2의 결과를 더해 생성



<그림 11> SHA-512의 각 라운드 구조

SHA-512

- 압축 함수 연산 (7/9)
  - 80 라운드 연산 (4/4)
    - 라운드 구조 (2/2)
      - 혼합기 (Mixer)
        - 버퍼 A~H를 갱신하기 위한 핵심 연산 블록
        - Mixer 1
          - A, B, C를 이용한 주함수 결과와 A의 순환 연산 결과를 결합
        - Mixer 2
          - E, F, G를 이용한 조건부 함수 결과, E의 순환 연산결과, 라운드 상수 K_t와 메시지 워드 W_t를 결합

Majority (x, y, z)
$(x \text{ AND } y) \oplus (y \text{ AND } z) \oplus (z \text{ AND } x)$

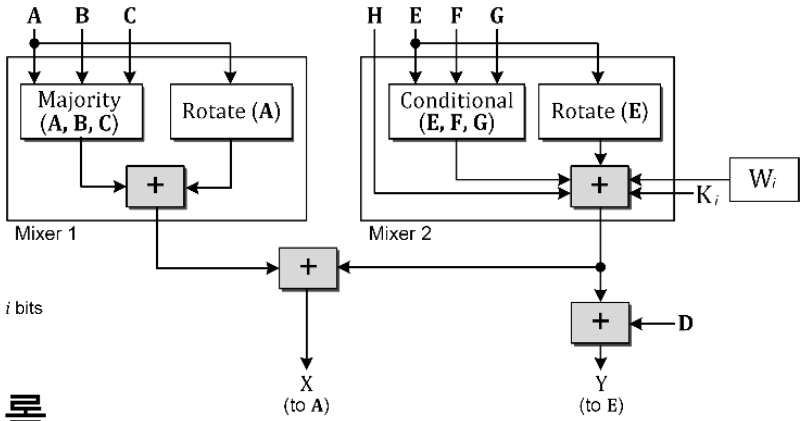
Rotate (A)
$RotR_{28}(A) \oplus RotR_{34}(A) \oplus RotR_{39}(A)$

Rotate (E)
$RotR_{14}(E) \oplus RotR_{18}(E) \oplus RotR_{41}(E)$

Conditional (x, y, z)
$(x \text{ AND } y) \oplus (\text{NOT } x \text{ AND } z)$

$+$ addition modulo 2^{64}

$RotR_i(x)$: Right-rotation of the argument x by i bits



<그림 11> SHA-512의 각 라운드 구조

- 최종 가산 수행
  - 80 라운드 종료 후 생성된 값에 이전 중간 다이제스트 값을 덧셈 연산
  - 마지막 블록에 대해 최종 가산을 수행한 중간 다이제스트가 최종 다이제스트

SHA-512

- 압축 함수 연산 (8/9)
  - 예제 12.7

주 함수를 버퍼 A, B, C에 적용한다. 만약 이 버퍼의 값 중 가장 왼쪽에 나타나는 16진수 수들이 각각 0x7, 0xA, 0xE라고 한다면 결과로 나오는 값의 가장 왼쪽에 나타나는 수는 무엇인가?

- 해당 16진수를 2진수로 변경
  - 0x7 = 0111, 0xA = 1010, 0xE=1110
- 각 수의 첫번째 비트는 0(0111), 1(1010), 1(1110) 이므로 주 함수에 의해 1
  - $(0 \text{ AND } 1) \oplus (1 \text{ AND } 1) \oplus (1 \text{ AND } 0) = 0 \oplus 1 \oplus 0 = 1$
- 각 수의 두번째 비트는 1(0111), 0(1010), 1(1110) 이므로 주 함수에 의해 1
  - $(1 \text{ AND } 0) \oplus (0 \text{ AND } 1) \oplus (1 \text{ AND } 1) = 0 \oplus 0 \oplus 1 = 1$
- 각 수의 세번째 비트는 1(0111), 1(1010), 1(1110) 이므로 주 함수에 의해 1
  - $(1 \text{ AND } 1) \oplus (1 \text{ AND } 1) \oplus (1 \text{ AND } 1) = 1 \oplus 1 \oplus 1 = 1$
- 각 수의 네번째 비트는 1(0111), 0(1010), 0(1110) 이므로 주 함수에 의해 0
  - $(1 \text{ AND } 0) \oplus (0 \text{ AND } 0) \oplus (0 \text{ AND } 1) = 0 \oplus 0 \oplus 0 = 0$

∴ 각 비트를 종합하면 1110 (16진수로는 0xE)

SHA-512

- 압축 함수 연산 (9/9)
  - 예제 12.8

조건부 함수를 E, F와 G 버퍼에 적용해보자. 만약 이 3개의 버퍼에서 가장 왼쪽에 나타나는 16진수 값들이 각각 0x9, 0xA와 0xF라고 한다면 결과로 값의 가장 왼쪽에 나타나는 값은 무엇인가?

- 해당 16진수를 2진수로 변경
  - 0x9 = 1001, 0xA = 1010, 0xF=1111
- 각 수의 첫번째 비트는 1(1001), 1(1010), 1(1111) 이므로 조건부 함수에 의해 1
  - $(1 \text{ AND } 1) \oplus (\text{NOT } 1 \text{ AND } 1) = 1 \oplus 0 = 1$
- 각 수의 두번째 비트는 0(1001), 0(1010), 1(1111) 이므로 조건부 함수에 의해 1
  - $(0 \text{ AND } 1) \oplus (\text{NOT } 0 \text{ AND } 1) = 0 \oplus 1 = 1$
- 각 수의 세번째 비트는 0(1001), 1(1010), 1(1111) 이므로 조건부 함수에 의해 1
  - $(0 \text{ AND } 1) \oplus (\text{NOT } 0 \text{ AND } 1) = 0 \oplus 1 = 1$
- 각 수의 네번째 비트는 1(1001), 0(1010), 1(1111) 이므로 조건부 함수에 의해 0
  - $(1 \text{ AND } 0) \oplus (\text{NOT } 1 \text{ AND } 1) = 0 \oplus 0 = 0$

∴ 각 비트를 종합하면 1110 (16진수로는 0xE)

목 차

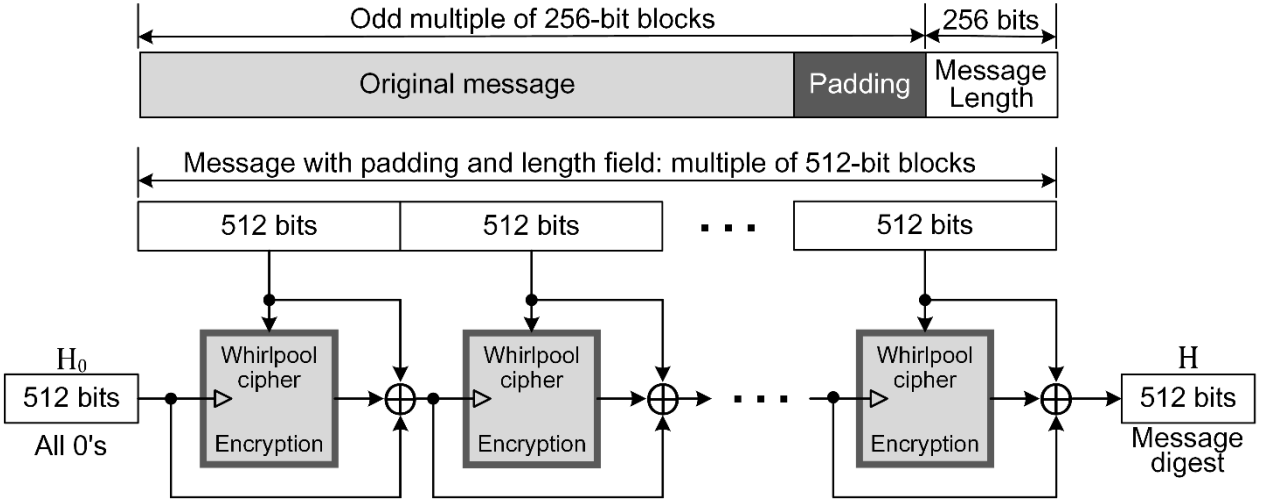
- 개요
- SHA-512
- WHIRLPOOL

WHIRLPOOL

- 개요

*NESSIE (New European Schemes for Signatures, Integrity, and Encryption)
*AES (Advanced Encryption Standard)

  - 512비트 메시지 다이제스트를 생성하는 암호학적 해시함수
  - 유럽 NESSIE에서 승인
  - Miyaguchi-Preneel 구조 기반
  - 512비트 메시지 블록 단위 처리
  - 512비트 초기값 H_0을 0으로 초기화하여 사용
  - AES를 수정한 Whirlpool 전용 블록 암호를 압축 함수로 사용



<그림 12> Whirlpool 해시함수

WHIRLPOOL

- 동작 과정

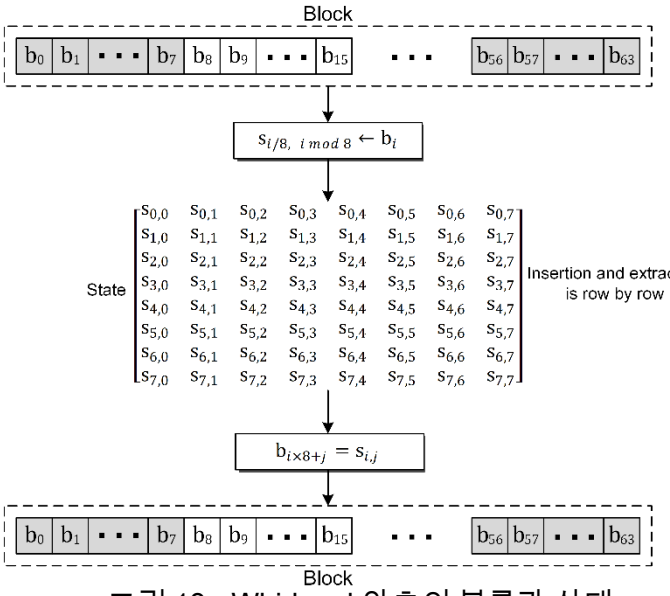
1. 입력 메시지 전처리
   - 전체 메시지를 512비트 메시지 블록으로 분할
   - 초기값 H_0를 0으로 설정
2. Whirlpool 암호 연산
   - 현재 메시지 블록을 Plaintext로 사용
   - 키 확장 후 10라운드 연산 수행
3. Miyaguchi-Preneel 결합
   - 결합 결과를 다음 메시지 블록 처리 시 Cipher Key로 사용
4. 모든 메시지 블록에 대해 2번부터 3번 과정을 반복

WHIRLPOOL

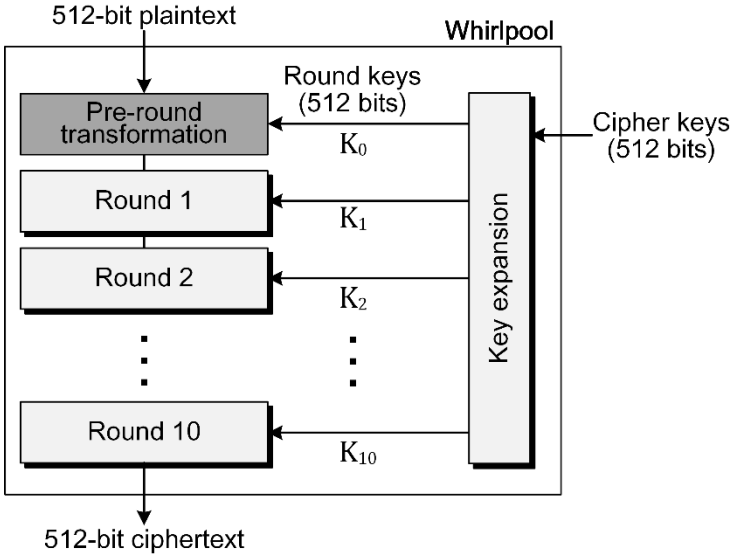
- 입력 메시지 전처리
  - 메시지 최대 길이 확인
    - 최대 2^{256}비트 메시지 처리 가능
  - 패딩
    - 첫번째 패딩 비트는 1, 나머지는 0으로 구성
    - 패딩 후 메시지 길이가 256비트의 홀수 배수가 되도록 조정
      - 길이 필드로 256비트가 추가되면 512비트의 배수가 되도록 패딩
  - 길이 필드 추가
    - 패딩된 메시지 뒤에 256비트 길이 필드 추가
    - 길이 필드에는 원래 메시지의 길이 저장
  - 초기값 H_0 설정
    - 초기값 H_0은 512 비트
    - 모든 비트를 0으로 설정

WHIRLPOOL

- Whirlpool 암호 연산 (1/9)
  - 구성 요소
    - 상태(State)와 블록
      - 블록과 상태의 크기가 512비트
      - 하나의 블록은 64바이트로 된 행 매트릭스로 간주
      - 상태는 8×8 정방 행렬로 간주
      - 블록-상태 혹은 상태-블록 변환을 행별로 수행
    - 라운드
      - 10 라운드로 구성
      - 블록과 키의 길이는 512비트
      - 각 길이가 512비트인 11개의 라운드 키로 K_0부터 K_{10} 사용



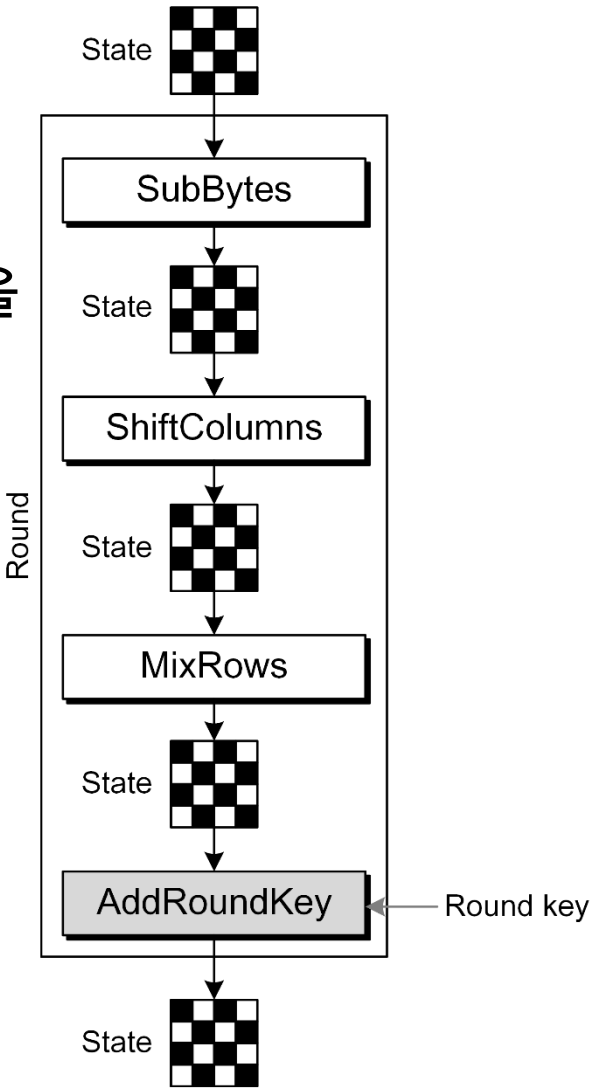
<그림 13> Whirlpool 암호의 블록과 상태



<그림 14> Whirlpool 암호의 일반적 아이디어

WHIRLPOOL

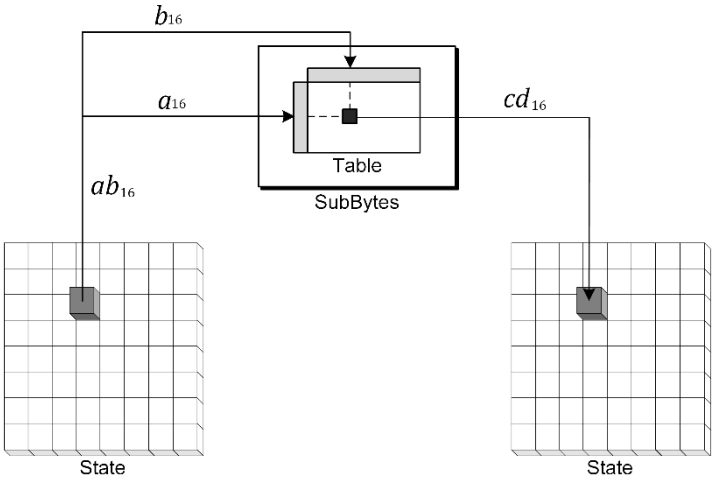
- Whirlpool 암호 연산 (2/9)
  - 라운드 구조 (1/6)
    - SubBytes
      - S-Box를 이용해 상태 행렬의 각 바이트 값을 다른 값으로 치환
    - ShiftColumns
      - 상태 행렬의 각 열을 정해진 크기만큼 아래로 순환 이동
    - MixRows
      - 각 행의 바이트를 수학적으로 혼합하여 행 내부의 확산 효과 생성
    - AddRoundKey
      - 현재 상태 값과 라운드 키를 XOR하여 키 정보를 반영



<그림 15> Whirlpool 암호의 각 라운드 구조

WHIRLPOOL

- Whirlpool 암호 연산 (3/9)
  - 라운드 구조 (2/6)
    - SubBytes (1/2)
      - 8×8 바이트 상태 행렬의 각 바이트를 S-Box를 통해 다른 바이트로 치환하는 과정
      - 하나의 바이트는 2개의 16진수로 표현
        - 왼쪽 16진수: S-Box의 행
        - 오른쪽 16진수: S-Box의 열
      - 각 바이트는 위치 변화 없이 값만 독립적으로 변환
      - 총 64개의 바이트를 독립적으로 치환



<그림 16> Whirlpool 암호의 SubBytes 변환

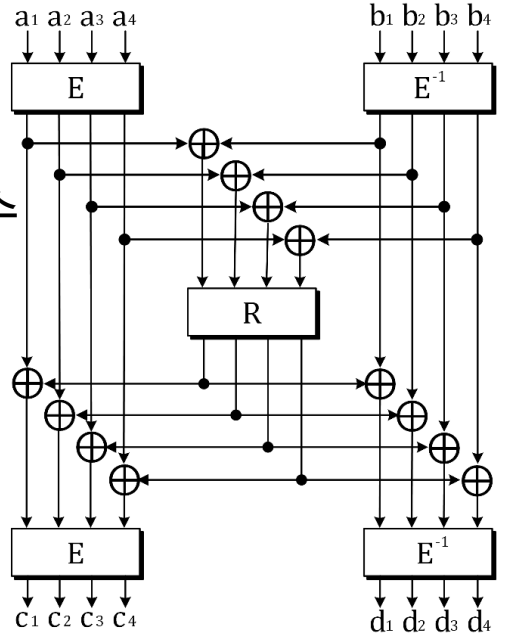
WHIRLPOOL

- Whirlpool 암호 연산 (4/9)
  - 라운드 구조 (3/6)
    - SubBytes (2/2)
      - S-Box 구성
        - 8비트 입력 바이트를 4비트 단위로 분리
        - 미니박스와 XOR연산을 통해 8바이트 출력 바이트 생성
        - 미니박스 E, E^{-1}, R을 조합하여 구성
          - E 미니박스: $GF(2^4)$ 기반 지수 연산 수행
          - E^{-1} 미니박스: E의 역함수
          - R 미니박스: 의사난수 생성기를 이용한 4비트 치환 함수

[표 8] Whirlpool의 S-Box

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	18	23	C6	E8	87	B8	01	4F	36	A6	D2	F5	79	6F	91	52
1	16	BC	9B	8E	A3	0C	7B	35	1D	E0	D7	C2	2E	4B	FE	57
2	15	77	37	E5	9F	F0	4A	CA	58	C9	29	0A	B1	A0	6B	85
3	BD	5D	10	F4	CB	3E	05	67	E4	27	41	8B	A7	7D	95	C8
4	FB	EE	7C	66	DD	17	47	9E	CA	2D	BF	07	AD	5A	83	33
5	63	02	AA	71	C8	19	49	C9	F2	E3	5B	88	9A	26	32	B0
6	E9	0F	D5	80	BE	CD	34	48	FF	7A	90	5F	20	68	1A	AE
7	B4	54	93	22	64	F1	73	12	40	08	C3	EC	DB	A1	8D	3D
8	97	00	CF	2B	76	82	D6	1B	B5	AF	6A	50	45	F3	30	EF
9	3F	55	A2	EA	65	BA	2F	C0	DE	1C	FD	4D	92	75	06	8A
A	B2	E6	0E	1F	62	D4	A8	96	F9	C5	25	59	84	72	39	4C
B	5E	78	38	8C	C1	A5	E2	61	B3	21	9C	1E	43	C7	FC	04
C	51	99	6D	0D	FA	DF	7E	24	3B	AB	CE	11	8F	4E	B7	EB
D	3C	81	94	F7	B9	13	2C	D3	E7	6E	C4	03	56	44	7F	A9
E	2A	BB	C1	53	DC	0B	9D	6C	31	74	F6	46	AC	89	14	E1
F	16	3A	69	09	70	B6	C0	ED	CC	42	98	A4	28	5C	F8	86

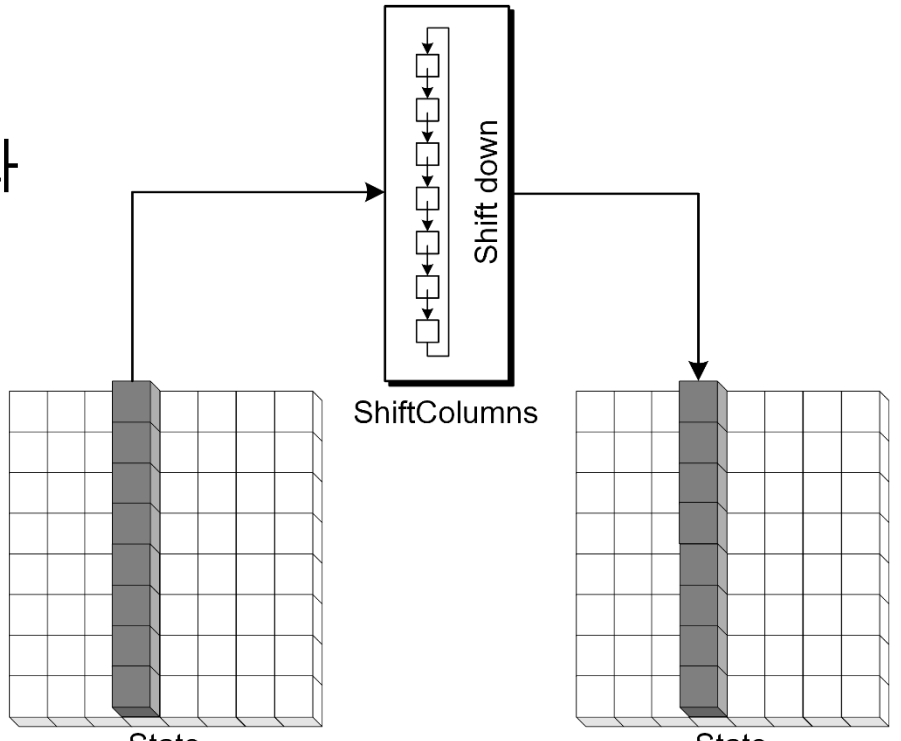
Input	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
Output	1	B	9	C	D	6	F	3	E	8	7	4	A	2	5	0	E box
Input	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
Output	F	0	D	7	B	E	5	A	9	2	C	1	3	4	8	6	E^{-1} box
Input	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
Output	7	C	B	D	E	4	9	F	6	3	8	A	2	5	1	0	R box



<그림 17> Whirlpool 암호의 SubBytes

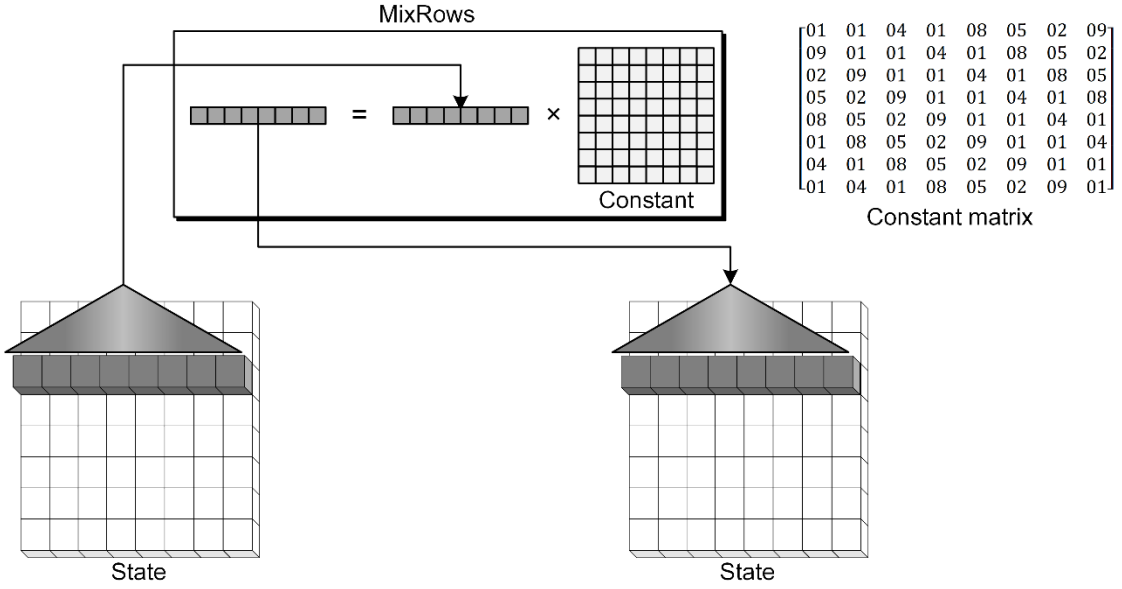
WHIRLPOOL

- Whirlpool 암호 연산 (5/9)
  - 라운드 구조 (4/6)
    - ShiftColumns
      - 열 단위 순환 이동 변환
      - 열의 위치마다 이동 방식의 차이 존재
        - 열 0은 0바이트 이동 (이동 없음)
        - 열 7은 7바이트 이동
      - 바이트 값 변화 없이 위치만 변화



<그림 18> Whirlpool 암호의 ShiftColumns 변환

- Whirlpool 암호 연산 (6/9)
  - 라운드 구조 (5/6)
    - MixRows
      - 행 단위 혼합 변환으로 확산 효과 발휘
      - 각 행을 상수 행렬과 곱하여 새로운 행으로 변환
      - 바이트 연산은 $GF(2^8)$ 위에서 수행
        - (0x11D) 또는 $(x^8 + x^4 + x^3 + x^2 + 1)$을 모듈로로 사용



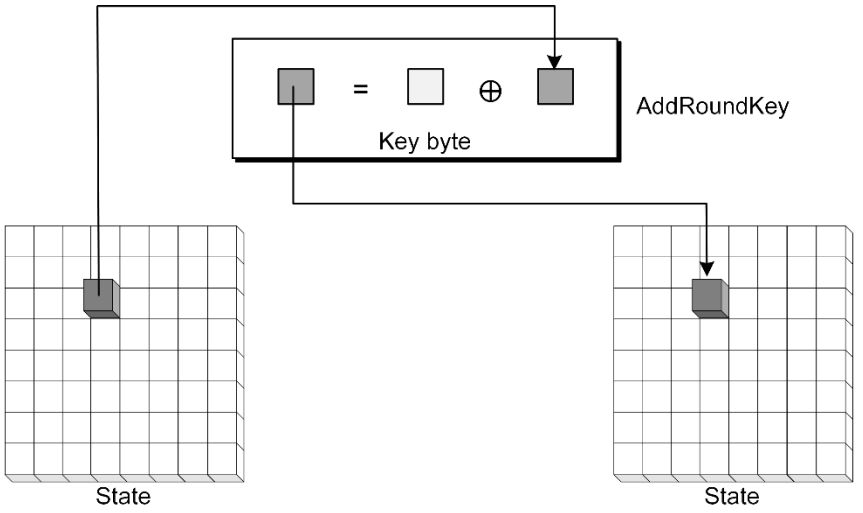
01	01	04	01	08	05	02	09
09	01	01	04	01	08	05	02
02	09	01	01	04	01	08	05
05	02	09	01	01	04	01	08
08	05	02	09	01	01	04	01
01	08	05	02	09	01	01	04
04	01	08	05	02	09	01	01
01	04	01	08	05	02	09	01

Constant matrix

<그림 19> Whirlpool 암호의 MixRows 변환

WHIRLPOOL

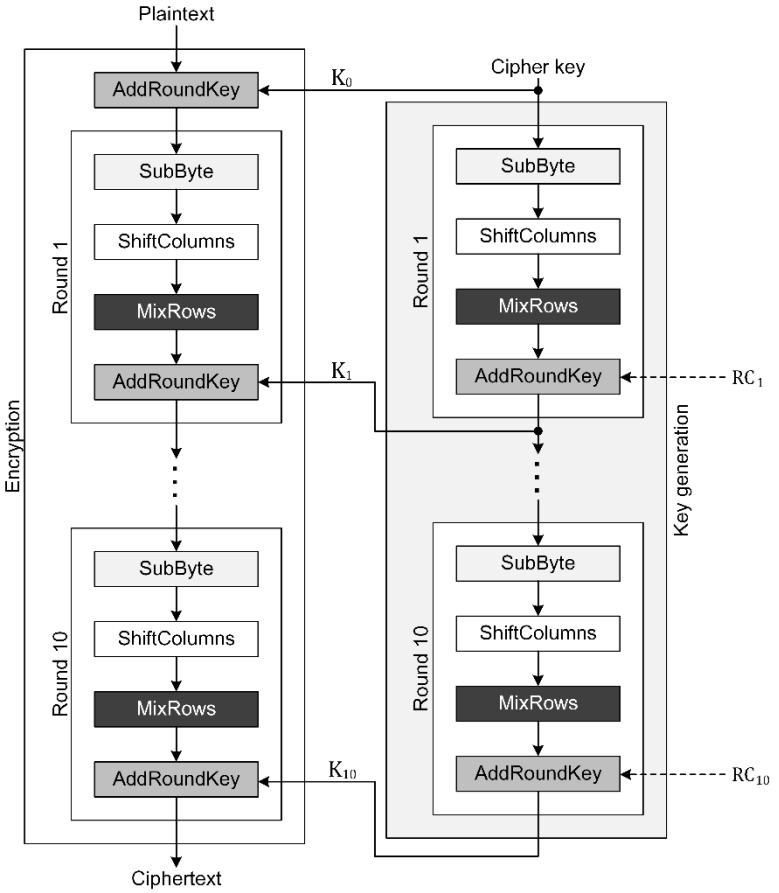
- Whirlpool 암호 연산 (7/9)
  - 라운드 구조 (6/6)
    - AddRoundKey
      - 라운드 키를 상태 행렬에 결합하는 변환
        - 상태 행렬과 라운드 키 모두 8×8 바이트 행렬
      - 대응되는 위치의 바이트 끼리 XOR 연산 수행
        - $GF(2^8)$의 덧셈은 8비트 워드의 XOR 연산과 동일
      - XOR 결과가 새로운 상태 행렬의 바이트 값



<그림 20> Whirlpool 암호의 AddRoundKey 변환

WHIRLPOOL

- Whirlpool 암호 연산 (8/9)
  - 키 확장 (1/2)
    - 별도의 키 확장 알고리즘 대신 라운드 함수 자체를 활용
    - 초기 암호 키 K_0에 라운드 함수를 반복 적용하여 $K_1 \sim K_{10}$ 생성
      - i번째 라운드 함수의 출력이 K_i
      - 실제 라운드 키 자리에 라운드 상수 $RC_1 \sim RC_{10}$을 가상 라운드 키로 사용



<그림 21> Whirlpool 암호의 키 확장

- Whirlpool 암호 연산 (9/9)
  - 키 확장 (2/2)
    - 라운드 상수 (RC, Round Constant)
      - 키 확장에 사용하는 10개의 상수 $\mathrm{RC}_1 \sim \mathrm{RC}_{10}$
      - 각 상수는 첫 번째 행만 0이 아닌 값을 갖는 8×8 행렬
        - 각 상수 행렬의 첫 번째 행의 값은 SubBytes 치환표(S-Box)에서 구성
          - $RC_{round}[row, column] = SubBytes\,(8(round-1)+column) \quad if\ row = 0$
          - $RC_{round}[row, column] = 0 \qquad\qquad if\ row \neq 0$

[표 8] Whirlpool의 S-Box

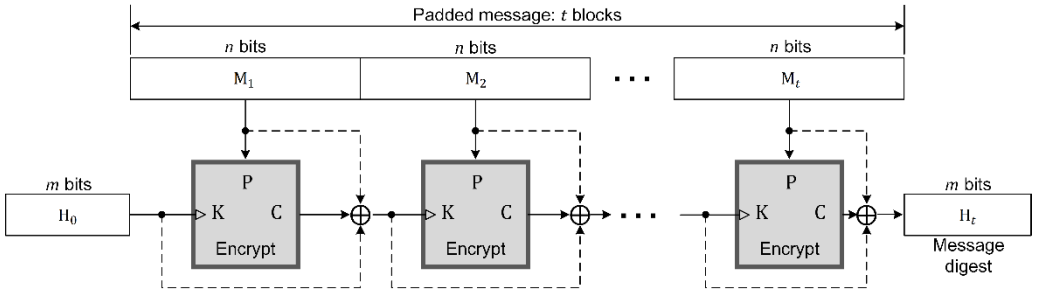
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	18	23	C6	E8	87	B8	01	4F	36	A6	D2	F5	79	6F	91	52
1	16	BC	9B	8E	A3	0C	7B	35	1D	E0	D7	C2	2E	4B	FE	57
2	15	77	37	E5	9F	F0	4A	CA	58	C9	29	0A	B1	A0	6B	85
3	BD	5D	10	F4	CB	3E	05	67	E4	27	41	8B	A7	7D	95	C8
4	FB	EE	7C	66	DD	17	47	9E	CA	2D	BF	07	AD	5A	83	33
5	63	02	AA	71	C8	19	49	C9	F2	E3	5B	88	9A	26	32	B0
6	E9	0F	D5	80	BE	CD	34	48	FF	7A	90	5F	20	68	1A	AE
7	B4	54	93	22	64	F1	73	12	40	08	C3	EC	DB	A1	8D	3D
8	97	00	CF	2B	76	82	D6	1B	B5	AF	6A	50	45	F3	30	EF
9	3F	55	A2	EA	65	BA	2F	C0	DE	1C	FD	4D	92	75	06	8A
A	B2	E6	0E	1F	62	D4	A8	96	F9	C5	25	59	84	72	39	4C
B	5E	78	38	8C	C1	A5	E2	61	B3	21	9C	1E	43	C7	FC	04
C	51	99	6D	0D	FA	DF	7E	24	3B	AB	CE	11	8F	4E	B7	EB
D	3C	81	94	F7	B9	13	2C	D3	E7	6E	C4	03	56	44	7F	A9
E	2A	BB	C1	53	DC	0B	9D	6C	31	74	F6	46	AC	89	14	E1
F	16	3A	69	09	70	B6	C0	ED	CC	42	98	A4	28	5C	F8	86

$$RC_3 = \begin{bmatrix} \mathbf{1D} & \mathbf{E0} & \mathbf{D7} & \mathbf{C2} & \mathbf{2E} & \mathbf{4B} & \mathbf{FE} & \mathbf{57} \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \end{bmatrix}$$

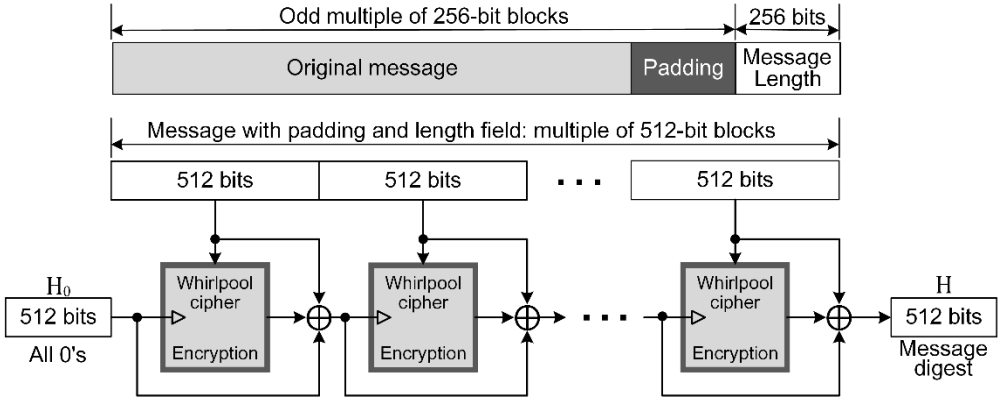
<그림 22> 라운드 3용 라운드 상수

WHIRLPOOL

- Miyaguchi-Preneel 결합 수행
  - 암호화 결과, 현재 메시지 블록, 이전 중간 다이제스트를 XOR 연산 수행
  - 결합 결과를 다음 메시지 블록 처리 시 Cipher Key로 사용



<그림 5> Miyaguchi-Preneel 구조



<그림 12> Whirlpool 해시함수

Thanks!

송 민 희(minhee@pel.sejong.ac.kr)

부록#1

- 암호학적 해시함수의 기준 (1/9)
  - 역상 저항성(Preimage Resistance) (1/2)
    - 정의
      - 어떤 메시지 M에 대한 해시 값 $h(M)$만을 가지고 메시지 M을 찾기 어렵게 하는 성질
    - 역상 공격
      - 주어진 정보 : $h(M)$
      - 구하려고 하는 것 : $h(M)$을 만족하는 M

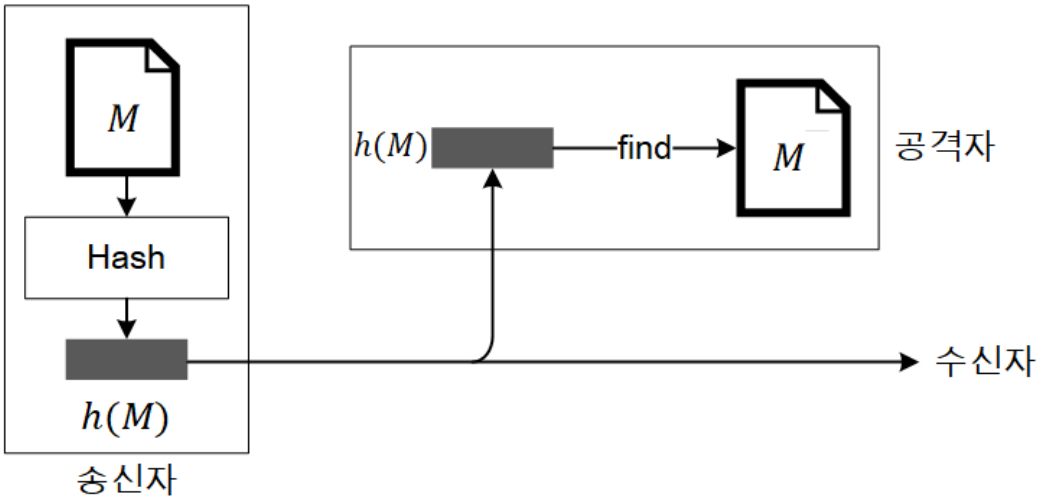
M : Message

Hash : Hash function

$h(M)$: Digest

Given : $h(M)$

Find : any M such that $h(M)$



부록#1

- 암호학적 해시함수의 기준 (2/9)
  - 역상 저항성(Preimage Resistance) (2/2)
    - 역상 저항성을 충족하지 못하는 예시

```
1.  def simple_hash(s):
2.      return sum(ord(c) for c in s) % 100
3.  M = “PEL”
4.  h_M = simple_hash(M)
```

```
5.  #역상 공격
6.  target_hash = h_M
7.  found = False
8.  for a in range(69, 91):
9.      for b in range(65, 91):
10.         for c in range(65, 91):
11.             s = chr(a) + chr(b) + chr(c)
12.             if simple_hash(s) == target_hash:
13.                 print(f"  → h(M) = {target_hash} 를 만족하는 M = '{s}' 발견")
14.                 found = True
15.                 break
16.         if found:
17.             break
18.     if found:
19.         break
```

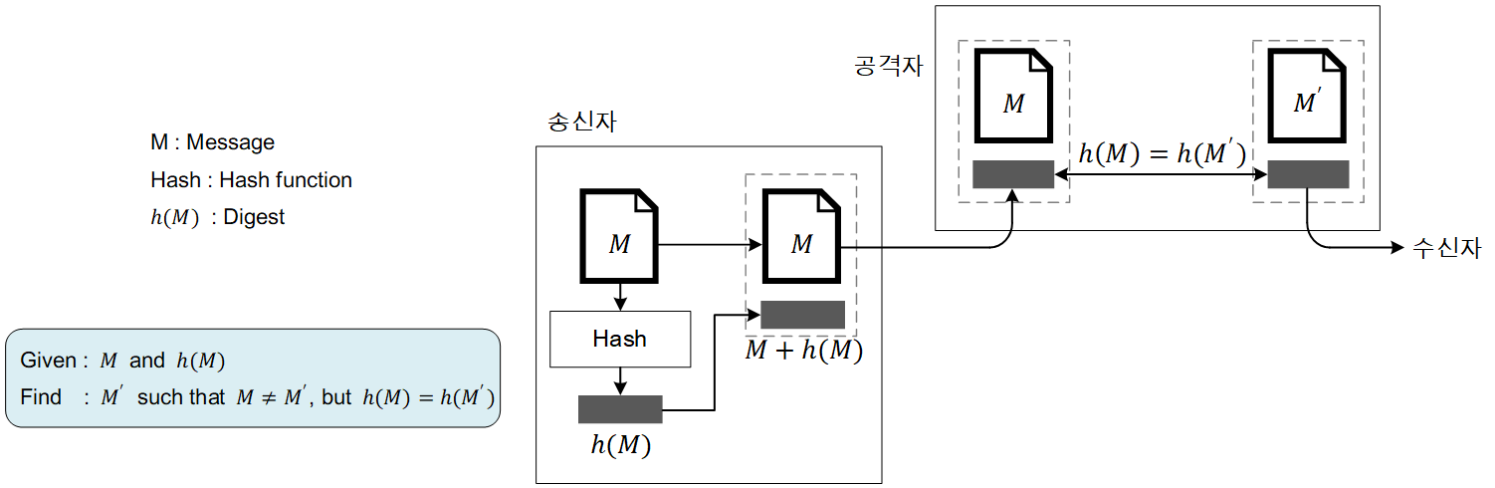
```
1. 프리이미지 공격
  → h(M) = 25 를 만족하는 M = 'AFZ' 발견
```

부록#1

- 암호학적 해시함수의 기준 (3/9)
  - 역상 저항성(Preimage Resistance) (2/2)
    - 역상 저항성을 충족하지 못하는 예시
      - checksum 함수
        - 데이터를 단순 수학 연산으로 요약한 값
        - 주어진 데이터의 checksum 값과 동일한 checksum 값을 갖는 여러 개의 다른 메시지를 만들 수 있으므로 역상 저항성을 충족하지 못함
      - StuffIt
        - Smith Micro Software에서 개발한 무손실 파일 압축 프로그램
        - 내부 무결성 검사에 checksum, CRC(Cyclic Redundancy Check)기반 방식을 사용하므로 동일한 결과를 만드는 입력을 쉽게 조작하거나 유추 할 수 있어 역상 저항성을 충족하지 못함

부록#1

- 암호학적 해시함수의 기준 (4/9)
  - 제 2 역상 저항성(Second Preimage Resistance)
    - 정의
      - 어떤 메시지 M과 해시 값$h(M)$이 있을 때, $h(M) = h(M')$을 만족하는 M'을 찾기 어렵게 하는 성질
    - 제 2 역상 공격
      - 주어진 정보 : M과 $h(M)$
      - 구하려고 하는 것 : $h(M) = h(M')$을 만족하는 $M'(\neq M)$



부록#1

- 암호학적 해시함수의 기준 (5/9)
  - 제 2 역상 저항성(Second Preimage Resistance)
    - 제 2 역상 저항성을 충족하지 못하는 예시

```
1.  def simple_hash(s):
2.      return sum(ord(c) for c in s) % 100
3.  M = “PEL”
4.  h_M = simple_hash(M)
```

```
2. 제2 프리이미지 공격
  → M = 'PEL', M′ = 'AFZ' (같은 해시 25)
```

```
5.  #제 2 역상 공격
6.  original = M
7.  original_hash = h_M
8.  found = False
9.  for a in range(65, 91):
10.     for b in range(65, 91):
11.         for c in range(65, 91):
12.             s = chr(a) + chr(b) + chr(c)
13.             if s != original and simple_hash(s) == original_hash:
14.                 print(f"  → M = '{original}', M′ = '{s}' (같은
    해시 {original_hash})")
15.                 found = True
16.                 break
17.         if found:
18.             break
19.     if found:
20.         break
```

부록#1

- 암호학적 해시함수의 기준 (6/9)
  - 제 2 역상 저항성(Second Preimage Resistance)
    - 제 2 역상 저항성을 충족하지 못하는 예시
      - MD5
        - 512비트 블록 단위로 데이터를 처리하고 128비트 해시 값을 출력하는 메시지 다이제스트 함수
        - 입력 데이터가 주어졌을 때, 동일한 해시 값을 갖는 다른 입력 값을 생성할 수 있으므로 제 2 역상 저항성을 충족하지 못함
      - RIPEMD-128
        - 512비트 블록으로 입력을 받고 128비트 해시 값을 출력하는 병렬 구조로 설계된 해시 함수
        - 출력 길이가 짧아 기존 해시 값과 같은 값을 만드는 다른 입력 값을 생성할 수 있으므로 제 2 역상 저항성을 충족하지 못함

부록#1

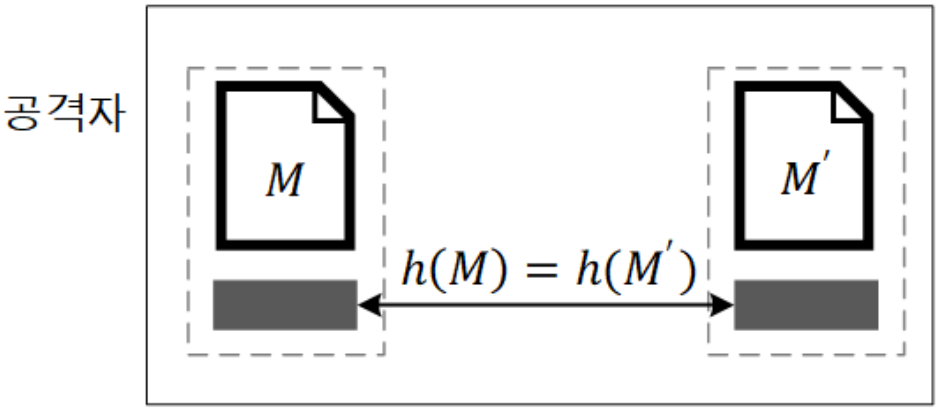
- 암호학적 해시함수의 기준 (7/9)
  - 충돌 저항성(Collision Resistance)
    - 정의
      - 서로 다른 두 메시지 M, M'에 대해서 $h(M) = h(M')$을 만족하는 임의의 2개의 입력 값(M, M')을 구하지 못하는 성질
    - 충돌 공격
      - 구하려고 하는 것 : $h(M) = h(M')$을 만족하는 쌍(M, M') (단, $M \neq M'$)

M : Message

Hash : Hash function

$h(M)$: Digest

Find : M and M' such that $M \neq M'$, but $h(M) = h(M')$



부록#1

- 암호학적 해시함수의 기준 (8/9)
  - 충돌 저항성(Collision Resistance)
    - 충돌 저항성을 충족하지 못하는 예시

```
1.  def simple_hash(s):
2.      return sum(ord(c) for c in s) % 100
3.  M = “PEL”
4.  h_M = simple_hash(M)
```

```
3. 충돌 공격
  → 충돌 발견! M = 'AAB', M′ = 'ABA', 해시값 = 96
```

```
5.  #충돌 공격
6.  hash_dict = {}
7.  found = False
8.  for a in range(65, 91):
9.      for b in range(65, 91):
10.         for c in range(65, 91):
11.             s = chr(a) + chr(b) + chr(c)
12.             h = simple_hash(s)
13.             if h in hash_dict:
14.                 print(f"  → 충돌 발견! M = '{hash_dict[h]}', M′ =
    '{s}', 해시값 = {h}")
15.                 found = True
16.                 break
17.             else:
18.                 hash_dict[h] = s
19.         if found:
20.             break
21.     if found:
22.         break
```

부록#1

- 암호학적 해시함수의 기준 (9/9)
  - 충돌 저항성(Collision Resistance)
    - 충돌 저항성을 충족하지 못하는 예시
      - SHA-1
        - 512비트 블록 단위로 메시지를 처리하고, 160 비트 해시 값을 출력하는 해시 함수
        - 짧은 해시 출력 길이와 구조적 결함으로 인해 서로 다른 두 입력이 같은 해시를 가질 수 있으므로 충돌 저항성을 충족하지 못함
      - HAVAL-128
        - 128 비트 해시 값을 출력하는 메시지 압축 기반 해시 함수
        - 설계 복잡도에 비해 출력 길이가 짧고 일부 라운드 설정에서 충돌 쌍이 계산적으로 쉽게 생성되므로 충돌 저항성을 충족하지 못함

부록#2

- Merkle-Damgård 구조 증명 (1/3)
  - 증명
    - 압축 함수 f가 충돌 저항성을 가지면, Merkle-Dagård 구조로 구성된 해시함수 H도 충돌 저항성을 가짐
  - 가정
    - 서로 다른 두 메시지 M, M'에 대해 $\mathrm{H}(\mathrm{M})=\mathrm{H}(\mathrm{M}')$이라고 가정
    - 메시지는 패딩 후 블록 단위로 분할됨
      - $\mathrm{M} \rightarrow \mathrm{M}_1, \mathrm{M}_2, \ldots, \mathrm{M}_\mathrm{t}$
      - $\mathrm{M}' \rightarrow \mathrm{M}'_1, \mathrm{M}'_2, \ldots, \mathrm{M}'_\mathrm{s}$
    - 각 중간 다이제스트는 다음과 같이 계산됨
      - $\mathrm{H}_0 = IV$
      - $\mathrm{H}_\mathrm{i} = f(\mathrm{H}_{\mathrm{i}-1}, \mathrm{M}_\mathrm{i})$

부록#2

- Merkle-Damgård 구조 증명 (2/3)
  - 증명 과정
    1. 서로 다른 메시지 M, M'이 같은 최종 해시값을 가진다고 가정
    2. 두 메시지는 서로 다르므로 패딩 후 생성된 블록열도 서로 다름
    3. 블록열과는 달리, 최종 다이제스트는 같으므로, 어떤 압축 함수 단계에서 서로 다른 입력이 같은 출력을 만든 지점이 존재
    4. 즉, 어떤 i, j에 대해 다음을 만족하는 경우가 발생
       - $f(H_{i-1}, M_i) = f\left(H'_{j-1}, M'_j\right)$ 단, $(H_{i-1}, M_i \neq H'_{j-1}, M'_j)$

       $\Rightarrow$ 어떤 압축 함수 f에서 충돌이 발생함

    $\therefore$ 전체 해시함수 H의 충돌을 찾을 수 있다면, 압축 함수 f의 충돌도 찾을 수 있음

부록#2

- Merkle-Damgård 구조 증명 (3/3)
  - 결론
    - 압축 함수 f가 충돌 저항성을 가진다면, f의 충돌을 찾는 것은 계산적으로 어려움
    - Merkel-Damgård 구조에서 H의 충돌은 f의 충돌로 이어지므로, H의 충돌을 찾는 것 또한 계산적으로 어려움

부록#3

- AES와 Whirlpool 비교
  - 기본 구조 비교

[표 9] AES와 Whirlpool 기본 구조 비교

특징	AES	Whirlpool
블록 크기	128비트	512비트
키 크기	128 / 192 / 256비트	512비트
상태 행렬	4×4 바이트 행렬	8×8 바이트 행렬
라운드 수	키 길이에 따라 10 / 12 / 14 라운드	10 라운드
연산 구성	SubBytes, ShiftRows, MixColumns, AddRoundKey	SubBytes, ShiftColumns, MixRows, AddRoundKey

  - 주요 연산 비교

[표 10] AES와 Whirlpool 주요 연산 비교

주요 연산	AES	Whirlpool
SubBytes	**4×4** 상태 행렬의 각 바이트를 **AES S-Box**로 치환	**8×8** 상태 행렬의 각 바이트를 **Whirlpool S-Box**로 치환
Shift-	ShiftRows: **행(row)을 왼쪽**으로 순환 이동	ShiftColumns: **열(column)을 아래쪽**으로 순환 이동
Mix-	MixColumns: 각 **열**을 $GF(2^8)$ 행렬 연산으로 혼합	MixRows: 각 **행**을 $GF(2^8)$ 행렬 연산으로 혼합
AddRoundKey	**4×4** 상태 행렬과 라운드 키를 XOR	**8×8** 상태 행렬과 라운드 키를 XOR
키 확장	**별도의 AES Key Schedule**로 라운드 키 생성	**라운드 함수 자체**를 활용해 K0~K10 생성

부록#3

- AES와 Whirlpool 비교
  - 기본 구조 비교

[표 9] AES와 Whirlpool 기본 구조 비교

특징	AES	Whirlpool
블록 크기	128비트	512비트
키 크기	128 / 192 / 256비트	512비트
상태 행렬	4×4 바이트 행렬	8×8 바이트 행렬
라운드 수	키 길이에 따라 10 / 12 / 14 라운드	10 라운드
연산 구성	SubBytes, ShiftRows, MixColumns, AddRoundKey	SubBytes, ShiftColumns, MixRows, AddRoundKey

  - 주요 연산 비교

[표 10] AES와 Whirlpool 주요 연산 비교

주요 연산	AES	Whirlpool
SubBytes	**4×4** 상태 행렬의 각 바이트를 **AES S-Box**로 치환	**8×8** 상태 행렬의 각 바이트를 **Whirlpool S-Box**로 치환
Shift-	ShiftRows: **행(row)을 왼쪽**으로 순환 이동	ShiftColumns: **열(column)을 아래쪽**으로 순환 이동
Mix-	MixColumns: 각 **열**을 $GF(2^8)$ 행렬 연산으로 혼합	MixRows: 각 **행**을 $GF(2^8)$ 행렬 연산으로 혼합
AddRoundKey	**4×4** 상태 행렬과 라운드 키를 XOR	**8×8** 상태 행렬과 라운드 키를 XOR
키 확장	**별도의 AES Key Schedule**로 라운드 키 생성	**라운드 함수 자체**를 활용해 K0~K10 생성