# FIT5037: Network Security
# Pseudorandom Number Generators and hash functions

Faculty of Information Technology
Monash University

MONASH University

# Copyright Notice



MONASH University

# Lecture 3: Pseudorandom Number Generators and hash functions

Lecture Topics:

- Symmetric key cryptography
- Asymmetric key cryptography
- **Pseudorandom Number Generators and hash functions**
- Authentication Methods and AAA protocols
- Security at Network layer
- Security at Network layer (continued)
- Security at Transport layer
- Security at Application layer
- Computer system security and malicious code
- Computer system vulnerabilities and penetration testing
- Intrusion detection
- Denial of Service Attacks and Countermeasures / Revision

MONASH University

# Outline

- Randomness and Random Number Generation
- NIST Recommended Constructions
- Hash functions and SHA-512
- Key Derivation Methods

MONASH University

# Randomness[a]

[a]Handbook of Applied Cryptography: Chapter 2

- Based on Probability Theory
- In PT an *experiment* is a procedure that yields a given set of outcomes
  - individual possible outcomes are called *simple events*
  - set of all possible outcomes is called *sample space*
  - for example the experiment of tossing a *fair* coin two times:
    - simple event: $\{H, T\}$
    - sample space: $\{HH, HT, TH, TT\}$
- A *probability distribution* $P$ on $S$ is a sequence of non-negative numbers $p_1, p_2, \ldots, p_n$ that sum to 1 where $p_i$ is interpreted as *probability of* $s_i$ being the outcome of the experiment
  - e.g. in the above coin toss experiment: $p_3 = \frac{1}{4}$ for the event $s_3 = TH$

MONASH University

# Randomness: Application in Security

- The basis of perfect security
  - A random key $k$ as long as the message $m$
    - probability is uniformly distributed
    - for $n$-bit key each bit has $\frac{1}{2}$ probability of being 0 and $\frac{1}{2}$ of being 1
  - ciphertext $c = k \oplus m$
  - not enough information for attacker (for any possible message $m$ there is a $k$ such that $c = m \oplus k$)
- Key generation for symmetric encryption
  - block ciphers such as AES
  - steam ciphers such as Counter and CHACHA20
- Nonce as a replay countermeasure or IV for modes of operation
- In challenge/response protocols for authentication
- Parameter generation of public key cryptography

MONASH University

# Random Number Generation

Two main strategies[1]:

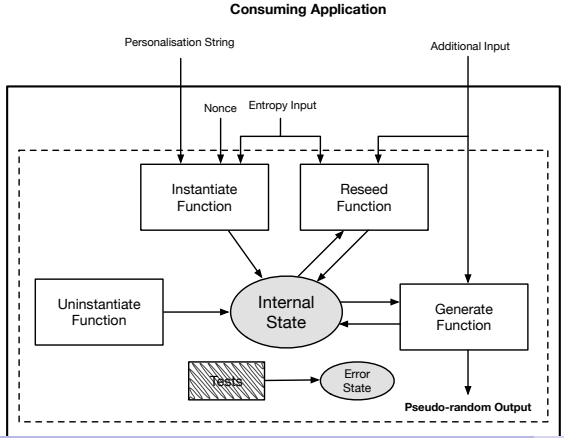
- Non-deterministic
  - based on an unpredictable physical process
- Deterministic
  - using algorithms with source of randomness
  - the output is referred to as pseudo-random
  - the source of randomness starts with a seed
    - the same bit sequence will be generated with the same seed

[1] NIST SP 800-90Ar: Recommendation for Random Number Generation Using Deterministic Random Bit Generators

# Pseudo-randomness or Deterministic RNGs [a]

[a] NIST SP 800-90Ar: Recommendation for Random Number Generation Using Deterministic Random Bit Generators

- Functional model of a Deterministic Random Bit Generator (DRBG)



# NIST DRBG

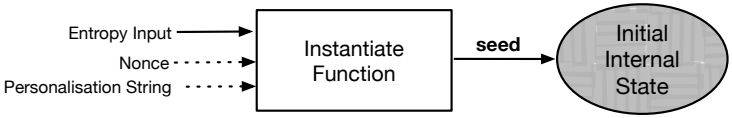
- Entropy Input
  - used to instantiate the internal state
  - used for seeding mechanism
  - both entropy input and (derived) seed must be kept secret
  - must meet security requirement of the DRBG mechanism
    - longer bit string can be provided to compensate for lack of adequate entropy
- Other Inputs
  - need not be secret (security of DRBG does not depend on secrecy of this)
  - can be used to verify validity
    - e.g. time (verify reasonable, format etc.)
  - Nonce maybe used with entropy input for instantiation
  - personalisation string
- Internal State
  - memory of the DRBG
    - parameters
    - variables and other stored data (e.g. security strength)
    - working state

MONASH University

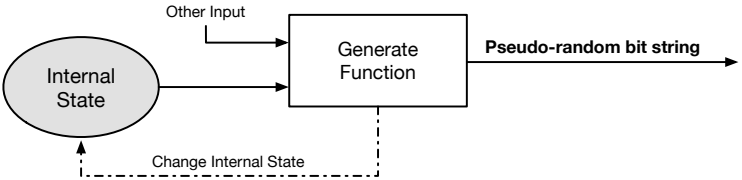
# NIST DRBG: Mechanism Functions

Five mechanism functions are defined:

1. Instantiate function:

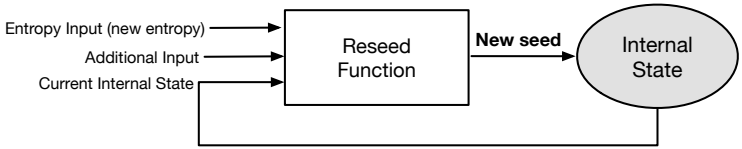



2. Generate function



# NIST DRBG: Mechanism Functions

3. Reseed function (optional)




4. Uninstantiate function: erases the internal state by zeroing the content
5. Health test function: checks that DRBG operates correctly

# NIST Hash_DRBG

- requires a hash function for: instantiation, reseed, and generate
- the hash function must meet the security requirement specified in NIST SP 800-57[2]

**Table 3: Hash functions that can be used to provide the targeted security strengths**

| Security Strength | Digital Signatures and hash-only applications | HMAC[22], Key Derivation Functions[23], Random Number Generation[24] |
|---|---|---|
| ≤ 80 | SHA-1[25] | |
| 112 | SHA-224, SHA-512/224, SHA3-224 | |
| 128 | SHA-256, SHA-512/256, SHA3-256 | SHA-1 |
| 192 | SHA-384, SHA3-384 | SHA-224, SHA-512/224 |
| ≥ 256 | SHA-512, SHA3-512 | SHA-256, SHA-512/256, SHA-384, SHA-512, SHA3-512 |

- uses a derivation function

[2]NIST SP 800-57: Recommendation for Key Management, Part 1: General

# NIST Hash_DRBG Internal State

- Consists of :
  1. `working_state`
     - variable `V` of length `seedlen`-bits: updated with each call to DRBG
     - constant `C` of length `seedlen`-bits: depends on the *seed*
     - `reseed_counter`: number of requests since new `entropy_input` (from instantiation or reseeding)
  2. administrative information
     - `security_strength`: of the DRBG instantiation
     - `prediction_resistance_flag`: whether prediction resistance capability is available
- Security of the DRBG depends on the values of `V` and `C`
  - these are the *secret values* of the internal state

MONASH University

# NIST Hash_df Derivation Function

- `Hash_df` has two inputs:
  - `input_string`: the string to be hashed
  - `no_of_bits_to_return`: represented as 32-bit integer and will be less than 255 $\times$ output length of the hash function
- Let:
  - `n=no_of_bits_to_return`
  - `outlen`: the output size of the hash function then
  - $len = \lceil \frac{n}{outlen} \rceil$

MONASH University

# NIST Hash_df Algorithm

```
Hash_df(input_string, no_of_bits_to_return)
1. temp = NULL
2. len = (no_of_bits_to_return // outlen) + 1
3. counter = 0x01
4. for i = 1 to len do
    4.1 temp = temp || Hash(counter || no_of_bits_to_return || input_string
    4.2 counter = counter + 1
5. requested_bits = leftmost(temp, no_of_bits_to_return)
6. return (SUCCESS, requested_bits)
```

MONASH University

# NIST Hash_DRBG Instantiate Algorithm

```
Hash_DRBG_Instantiate(entropy_input,nonce,personalisation_string,security_s

1. seed_material = entropy_input || nonce || personalisation_string
2. seed = Hash_df(seed_material, seedlen)
3. V = seed
4. C = Hash_df(0x00 || V, seedlen)
5. reseed_counter = 1
6. return (V, C, reseed_counter)
```

- The reseed function is the same with exception of:

```
1. seed_material = 0x01 || V || entropy_input || additional_input
```

MONASH University

# NIST Hash_DRBG Generate Algorithm

- working_state: V, C, reseed_counter

```
Hash_DRBG_Generate()
1. if reseed_counter > reseed_interval then do
    1.1 idicate reseed is required
2. if additional_input <> NULL then do
    2.1 w = Hash(0x02 || V || additional_input)
    2.2 V = (V + w) mod 2^seedlen
3. returned_bits = Hashgen(requested_number_of_bits, V)
4. H = Hash(0x03 || V)
5. V = (V + H + C + reseed_counter) mod 2^seedlen
6. reseed_counter = reseed_counter + 1
7. return (SUCCESS, returned_bits, V, C, reseed_counter)
```

MONASH University

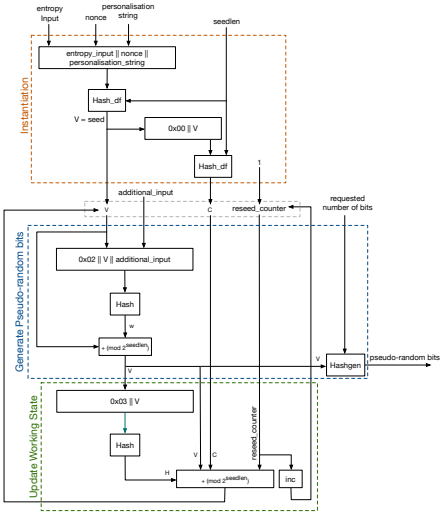
# NIST Hash_DRBG Hashgen algorithm

- n: requested number of bits

```
Hashgen(n, outlen)
1. m = (n // outlen) + 1
2. data = V
3. W = NULL
4. for i = 1 to m do
    4.1 w = Hash(data)
    4.2 W = W || w
    4.3 data = (data + 1) mod 2^seedlen
5. returned_bits = leftmost(W, n)
6. return (returned_bits)
```

MONASH University

# NIST Hash_DRBG Overall Design



# NIST Other Derivation Function

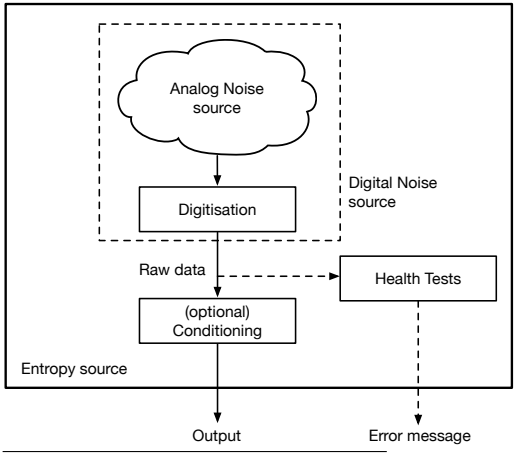
NIST SP 800-90Ar1 proposes additional derivation functions:

- HMAC based algorithms for instantiate, reseed and generate mechanisms
- DRBG mechanism based on block ciphers
  - uses counter mode of operation with an approved block cipher

MONASH University

# Non-deterministic Entropy Sources

- NIST SP 800-90B [3] provides the following entropy source model for non-deterministic RBGs




[3] NIST SP 800-90B: Recommendation for the Entropy Sources Used for Random Bit Generation

# NIST Entropy Source Model

- the *noise source* is the root of security for this model
  - if the sampling (of noise) does not produce binary data a digitisation process is needed
    - raw data
  - two categories of noise sources:
    - *physical noise source*: dedicated hardware to generate randomness
    - *Non-physical noise sources*: use system data such as output of API functions, RAM data, system time, human input (e.g. mouse movement)
- conditioning component: may be used to reduce bias
  - HMAC, CMAC and CBC-MAC are proposed for keyed conditioning
  - an approved hash function, `Hash_df` and `Block_Cipher_df` (in NIST SP-800-90A)
- health test:
  - to detect failures of the noise source
    - start-up tests
    - continuous tests
    - on-demand tests

MONASH University

# Examples of Continuous Health Tests

- **Repetition Count Test**
  - to detect if noise source has become stuck on a single output value for a long period of time
  - a cutoff value $C$ is calculated (based on probability of repetition and false-positive)
  - an error will be generated if a sample is repeated $C$ or more times
- **Adaptive Proportion Test**
  - to detect a large loss of entropy due to change affecting the noise source
  - measures the frequency of occurrence of a sample
  - an error will be generated if a sample occurs more frequently than expected (more subtle variations)
- **Permutation Testing**
  - NIST specifies 11 permutation tests (refer to the document for further detail[4])

[4] NIST SP 800-90B: Recommendation for the Entropy Sources Used for Random Bit Generation

# PRNG in Linux Kernel

- Linux Kernel provides two interfaces to its Pseudo-random number generation
  - `/dev/random`
  - `/dev/urandom`
- Kernel maintains an entropy pool gathering randomness from various sources
  - it also maintains an entropy estimate
    - `/dev/random` will block if there is not enough entropy in the pool
    - `/dev/urandom` is a non-blocking interface and returns requested number of bits regardless of available entropy
- for long-term keys such as public/private key pairs `/dev/random` should be used
  - `/dev/random` can also be used to generate seeding material for a well-designed PRNG

MONASH University

# Hash Functions: Overview

- Hash function is a *one-way* function that accepts a message M and produces a fixed-size message digest H(M)
- Everyone (including the adversary) knows the hash algorithm and can generate the message digest
- used with symmetric encryption to construct message authentication code
- used with asymmetric encryption to form digital signature

MONASH University

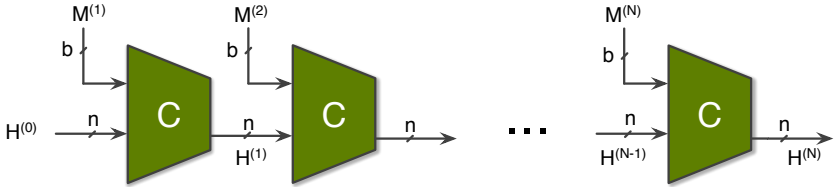
# Requirements of Cryptographic Hash Functions

- Can be applied to any sized message M
- Produces fixed-length output h
- Is easy to compute h=H(M) for any message M
- **One-way property** or **Pre-image resistance**:
  - Given h, it is computationally infeasible to find x such that H(x)=h (h is the image of x, x is the pre-image of h)
- **Weak Collision resistance** or **Second pre-image resistance**:
  - Given x, it is computationally infeasible to find $y \neq x$ such that H(y)=H(x)
- **Strong collision resistance** or **Collision resistance**:
  - It is computationally infeasible to find any pair (x,y) such that H(x)=H(y)

MONASH University

# SHA2: SHA-512

- Published in FIPS PUB 180-4: Secure Hash Standard[5]
- SHA Family General Design:

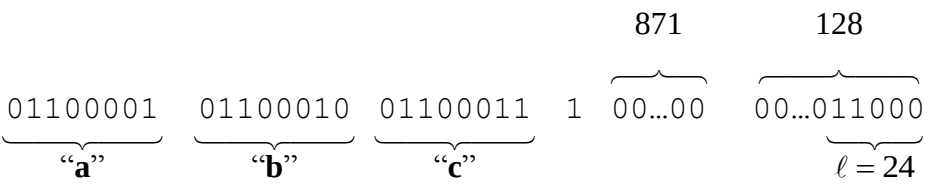



- SHA-512
  - Process the message in 1024-bit blocks.
  - Compression function performs 80 rounds of operations.
  - Each round takes as input the 1024-bit message block $M^{(i)}$, the 512-bit buffer value $H^{(i-1)}$, and updates the contents of that buffer as $H^{(i)}$.
    - $H^{(N)}$ is the final message digest

[5] FIPS PUB 180-4: Secure Hash Standard

# SHA-512 Padding

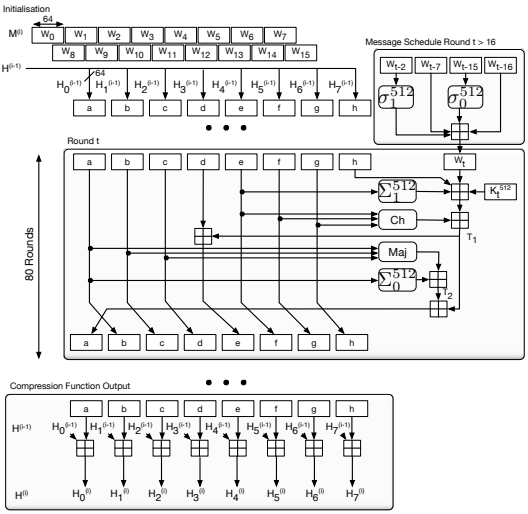
- All messages are padded as follows
  - $l$ specifies the size of message in bits
    - last 128 bits of padding
  - $\texttt{padding} = 1||0^k||l^{128}$
    - $l + 1 + k \equiv 896 \bmod 1024$ ($l + 1 + k + 128 \equiv 0 \bmod 1024$)
  - Example: the 8-bit ASCII message "abc"



# SHA-512 Initialisation

- 8 64-bit variables $a, b, c, d, e, f, g, h$ are set with $H^0$ defined as
  - $a = H_0^{(0)} = 6a09e667f3bcc908$
  - $b = H_1^{(0)} = bb67ae8584caa73b$
  - $c = H_2^{(0)} = 3c6ef372fe94f82b$
  - $d = H_3^{(0)} = a54ff53a5f1d36f1$
  - $e = H_4^{(0)} = 510e527fade682d1$
  - $f = H_5^{(0)} = 9b05688c2b3e6c1f$
  - $g = H_6^{(0)} = 1f83d9abfb41bd6b$
  - $h = H_7^{(0)} = 5be0cd19137e2179$
- Padded message is divided into 1024-bit blocks $M^{(1)}, M^{(2)}, \ldots, M^{(N)}$
  - $N = \dfrac{\text{Padded Message Length in bits}}{1024}$
  - The 1024-bit block $M^{(i)}$ input to compression function at step $i$ is copies into 16 64-bit word $W_0, W_1, \ldots, W_{15}$
  - a message schedule function expands these initial words through 80 rounds

# SHA-512 Compression Function




$$Ch(x,y,z) = (x \wedge y) \oplus (\neg x \wedge z)$$
$$Maj(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$

$$\sum_0^{\{512\}}(x) = ROTR^{28}(x) \oplus ROTR^{34}(x) \oplus ROTR^{39}(x)$$
$$\sum_1^{\{512\}}(x) = ROTR^{14}(x) \oplus ROTR^{18}(x) \oplus ROTR^{41}(x)$$
$$\sigma_0^{\{512\}}(x) = ROTR^{1}(x) \oplus ROTR^{8}(x) \oplus SHR^{7}(x)$$
$$\sigma_1^{\{512\}}(x) = ROTR^{19}(x) \oplus ROTR^{61}(x) \oplus SHR^{6}(x)$$

- where
  - $ROTR^n(x)$: Rotate $x$, $n$-bit to the right (circular right shift)
  - $SHR^n(x)$: Shift $x$, $n$-bit to the right (zeros entering from left)
  - $\wedge$: bitwise AND
  - $\vee$: bitwise OR
  - $\oplus$: bitwise XOR
  - $\neg$: bitwise complement
  - $\boxplus$: addition modulo $2^{64}$

# SHA-512 Round Constants

- Each round uses a 64-bit round constant $K_t^{(512)}$
  - constants are the first 64 bits of the fractional parts of the cube root of the first 80 prime numbers

```
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 597f299cfc657e2a 5fcb6fab3ad6faec 6c44198c4a475817
```

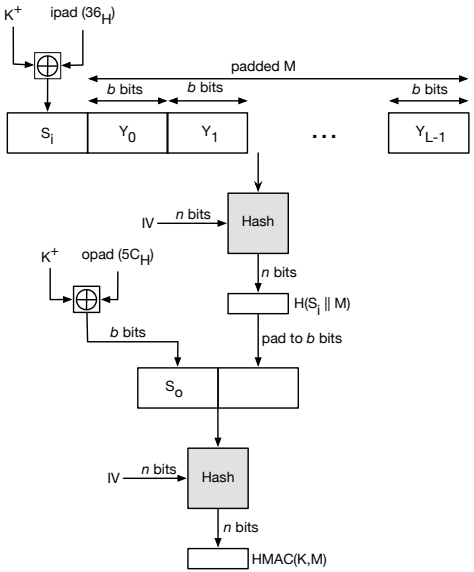
# HMAC (revision)

- specified as Internet standard RFC 2104[6]
- designed to
  - use, without modifications available hash functions
  - allow for easy replaceability of embedded hash function
  - preserve original performance of hash function without significant degradation
  - use and handle keys in a simple way.
- uses hash function on the message with a key:
  - $\text{HMAC}_K(M) = \text{Hash}\left[(K^+ \oplus \text{opad}) \,||\, \text{Hash}[(K^+ \oplus \text{ipad}) \,||\, M)]\right]$
  - where $K^+$ is the key padded with zero to $b$ bits (inner hash block size)
    - if $|K| > b$ then $K^+ = \text{Hash}[K]$
  - $ipad = 36^b$ and $opad = 5C^b$ are padding constants repeated to $b$ bits
- overhead is 2 additional input blocks
  - $S_i = K^+ \oplus 36^b$ added to the beginning of message
  - $S_o = K^+ \oplus 5C^b || H(S_i || M)$

[6] RFC2104 HMAC: Keyed-Hashing for Message Authentication

# HMAC Overview

- L: number of blocks in M
- b: number of bits in a block
- n: length of message digest produced by embedded hash
- K: secret key
- $K^+$: K padded with zeros on the left to become b bits in length
- ipad: 36 hex repeated b/8 times
- opad: 5C hex repeated b/8 times




MONASH University

# Key Derivation Methods

- Used to derive keys from shared secrets
  - expand keys
  - multiple usage e.g. symmetric encryption and message authentication code
- Various protocols require a KDM
  - TLS 1.3
  - IPSec
- NIST SP 800-56Cr1 provides recommendation for KDM[7]
  - One-Step
  - Two-Step

[7] NIST SP 800-56Cr1

MONASH University

# NIST One-Step KD

- Key Derivation Function (KDF) is executed in a single step
  - defines an auxiliary function $H$ which can be:
    - an approved hash function
    - HMAC with an approved hash
    - a variant of KMAC defined in NIST SP 800-185
  - when $H$ is a MAC then a known *salt* can be used as key
    - why is this allowed?
    - the MAC must satisfy the security requirement that with the key and a part of shared secret (input) known the complete output is unpredictable
  - inputs to the KDM:
    - shared secret from Kex
    - (an indication of) desired output bit length
    - other information (e.g. *salt* when KDF is a MAC)

MONASH University

# NIST One-Step KD

function call: `KDM(Z, OtherInput)`

- $Z$: shared secret from Kex
- `OtherInput`
  - $L < len(H\_outputBits) \times (2^{32} - 1)$: desired length of keying material in bits
    - $len(H\_outputBits)$: size of the HMAC output in this example (e.g. 512 in HMAC-SHA512)
  - *FixedInfo* : a constant context-specific data (e.g. the string "Key Derivation")
  - *salt*: optional when hash is used, necessary e.g. for HMAC
- `max_H_inputBits`: maximum message size for hash function H

MONASH University

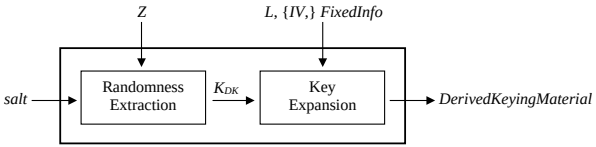
# NIST One-Step KD: Process Pseudocode

```
1. if L > 0 then reps = L // H_outputBits + 1
2. if reps > (2**32 - 1) then exit with error
3. counter = 0x00000000
4. if len(counter || Z || FixedInfo) > max_H_inputBits then exit with erro
5. Result[0] = NULL
6. for i = 1 to reps
    6.1 counter = counter + 1
    6.2 K[i] = H(counter || Z || FixedInfo)
    6.3 Result[i] = Result[i-1] || K[i]
7. return L leftmost bits of Result[reps]
```

MONASH University

# NIST Two-Step KD: Overview

- similar to one-step has $Z$, $L$, and `FixedInfo` as input
- in contrast to one-step requires a *salt*
- the method is referred to as *extraction-then-expansion* (shown below)




- extraction step:
  - HMAC or AES-CMAC can be used
  - `salt` is used as the key to the MAC and $Z$ as the message
  - the output is used as *Key Derivation Key* ($K_{DK}$)
    - $K_{DK}$ must be used only once as key to the expansion process
- key expansion step:
  - use $K_{DK}$, $L$ and other info with a PRF
  - approved PRFs: HMAC or CMAC

# SHA-3

- Published as FIPS PUB 202: SHA-3 Standard[8]
  - various functions based on KECCAK (winner of NIST competition)
  - also specifies KECCACK-p family of mathematical permutations
- Consists of four hash functions:
  - SHA3-224
  - SHA3-256
  - SHA3-384
  - SHA3-512
- Also specifies two extendable-output functions (XOFs)
  - SHAKE128
  - SHAKE256

[8] FIPS PUB 202: SHA-3 Standard

MONASH University

# References

Some of the algorithms and diagrams in the following references are reproduced in this document.


- Handbook of Applied Cryptography: Chapter 2
- NIST SP 800-90Ar: Recommendation for Random Number Generation Using Deterministic Random Bit Generators
- NIST SP 800-57: Recommendation for Key Management, Part 1: General
- NIST SP 800-90B: Recommendation for the Entropy Sources Used for Random Bit Generation
- FIPS PUB 180-4: Secure Hash Standard
- NIST SP 800-56Cr1: Recommendation for Key-Derivation Methods in Key-Establishment Schemes
- FIPS PUB 202: SHA-3 Standard
- RFC2104 HMAC: Keyed-Hashing for Message Authentication


MONASH University