



- In Tutorial 1, we saw an example about how large software such as a Linux distribution is typically distributed as an ISO image along with a SHA-256 (see Sec. 3.1) digest for integrity check.

On a Linux platform, which typically comes with the utility `sha256sum`, we can use this command to compute the SHA-256 digest of a file:

```
sha256sum -b filepath
```

On a contemporary Windows platform, which typically comes with the PowerShell cmdlet `Get-FileHash`, we can use this command to compute the SHA-256 digest of a file:

```
Get-FileHash filepath -Algorithm SHA256
```

We have also seen other applications of hash functions. For example,

- In the previous lecture (see [knowledge base entry](#)), we encountered the usage of a hash function in the LRW tweakable block cipher construction [LRW02].
- Hash functions can also be used to construct message authentication and authenticated encryption schemes. As such, we discuss hash functions before message authentication and authenticated encryption.

Let us now define:

#### Definition 1: Hash function [KL21, Definition 6.1]

A hash function with output length  $\ell(n)$  is a pair of PPT algorithms  $(\text{Gen}, H)$  where

- $\text{Gen}$  is a *probabilistic* algorithm that takes as input a security parameter  $1^n$  and outputs a key  $s$ .
- $H$  is a *deterministic* algorithm that takes as input **1** a key  $s$  and **2** a string  $x \in \{0, 1\}^*$ ; and outputs a string  $H^s(x) \in \{0, 1\}^{\ell(n)}$ .

If  $H^s$  is defined only for inputs of length  $\ell'(n) > \ell(n)$ , then we say that  $(\text{Gen}, H)$  is a fixed-length hash function for inputs of length  $\ell'(n)$ , and we call  $H$  a *compression function*.

- When using a hash function, we actually choose a hash function out of a family of hash functions using a key.
- We use the notation  $H^s$  instead  $H_s$  to highlight the fact that the key  $s$  does not need to be secret.
- Although in general a hash function takes a key, we focus on *unkeyed* hash functions (i.e., the key is fixed) in this lecture and will discuss keyed hash functions elsewhere.

An obvious requirement on hash functions is *collision-resistance*: it should be infeasible to find a *collision*, namely two inputs/preimages that produce the same digest; or formally,

#### Definition 2: Collision resistance [KL21, Definition 2]

The collision-finding experiment  $\text{Exp}_{\mathcal{A}, \mathcal{H}}^{\text{cr}}(n)$ :

1.  $\text{Gen}(1^n)$  generates a key  $s$ .

2. Adversary  $\mathcal{A}$  receives  $s$  and outputs  $x, x'$ .

3. The output of the experiment, denoted by  $\text{Exp}_{\mathcal{A}, \mathcal{H}}^{\text{cr}}(n)$ , is defined as

$$\text{Exp}_{\mathcal{A}, \mathcal{H}}^{\text{cr}}(n) \triangleq \begin{cases} 1 & \text{if } x \neq x' \text{ and } H^s(x) = H^s(x'), \\ 0 & \text{otherwise.} \end{cases}$$

A hash function  $\mathcal{H} = (\text{Gen}, H)$  is collision-resistant if for every PPT adversary  $\mathcal{A}$ , there is a negligible function  $\text{negl}$  such that

$$\Pr \left\{ \text{Exp}_{\mathcal{A}, \mathcal{H}}^{\text{cr}}(n) = 1 \right\} \leq \text{negl}(n).$$

- It has become a common practice to refer to  $H$  or  $H^s$  rather than  $\mathcal{H}$  as the hash function. The context of discussion is usually clear about whether “hash function” refers to  $\mathcal{H}$  or  $H$  or  $H^s$ .
- The existence of collision-resistant hash functions appears to be a qualitatively stronger assumption than the existence of other symmetric-key primitives, while at the same time being weaker than what is needed for public-key encryption.
- Collision-resistant hash functions have important applications in both the symmetric-key and public-key settings (see Sec. 2.1).

Hash functions used in practice are more often *unkeyed* than *keyed* (as defined in Definition 1).

- An unkeyed hash function is a deterministic function; an adversary can theoretically pre-compute a collision for each preimage, but fortunately, this pre-computation is impractical.
- A security proof for a scheme based on a collision-resistant hash function is still meaningful when an unkeyed hash function  $H$  is used, provided the proof shows that any efficient adversary breaking the scheme can efficiently find a collision in  $H$ .
- In general, unkeyed and keyed hash functions serve as candidates for *modification / manipulation detection codes* (MDCs) and *message authentication codes* (MACs) respectively.

For the rest of this lecture, we shall discuss

1. in Sec. 2, the security definitions/notions besides collision resistance for hash functions (which are different from the security definitions for *encryption* in Lecture 3);
2. in Sec. 3, a compression-based construction of hash functions called the Merkle-Damgård construction, on which the SHA-2 family of hash functions is based;
3. in Sec. 4, a permutation-based construction of hash functions called the Sponge construction, on which the SHA-3 family of hash functions is based;

## 2 Security notions

For some applications, security requirements weaker than collision resistance suffice:

**Second-preimage resistance:** Given  $s$  and a uniform  $x$ , it is infeasible for a PPT adversary to find  $x' \neq x$  such that  $H^s(x') = H^s(x)$ .

Collision resistance (as per Definition 2) is a stronger notion than second-preimage resistance, because in the former case, the adversary has the freedom to choose any pair of  $x$  and  $x'$ .

Collision resistance  $\implies$  second-preimage resistance.

**Preimage resistance:** Given  $s$  and  $y = H^s(x)$  for a uniform  $x$ , it is infeasible for a PPT adversary to find  $x'$  — either  $x' = x$  or  $x' \neq x$  — such that  $H^s(x') = y$ .  $\rightarrow$  A preimage-resistant  $H^s$  is so-called *one-way*.

Second-preimage resistance is a stronger notion than preimage resistance, because in the former case, the adversary has access to  $x$  (more information) rather than just an image of  $x$ .

Second-preimage resistance  $\implies$  preimage resistance.

An obvious application of the preceding notions is a simpler security check: a scheme that fails a weaker notion must also fail a stronger notion.

Figure 1 classifies the concepts, including security notions, introduced so far.

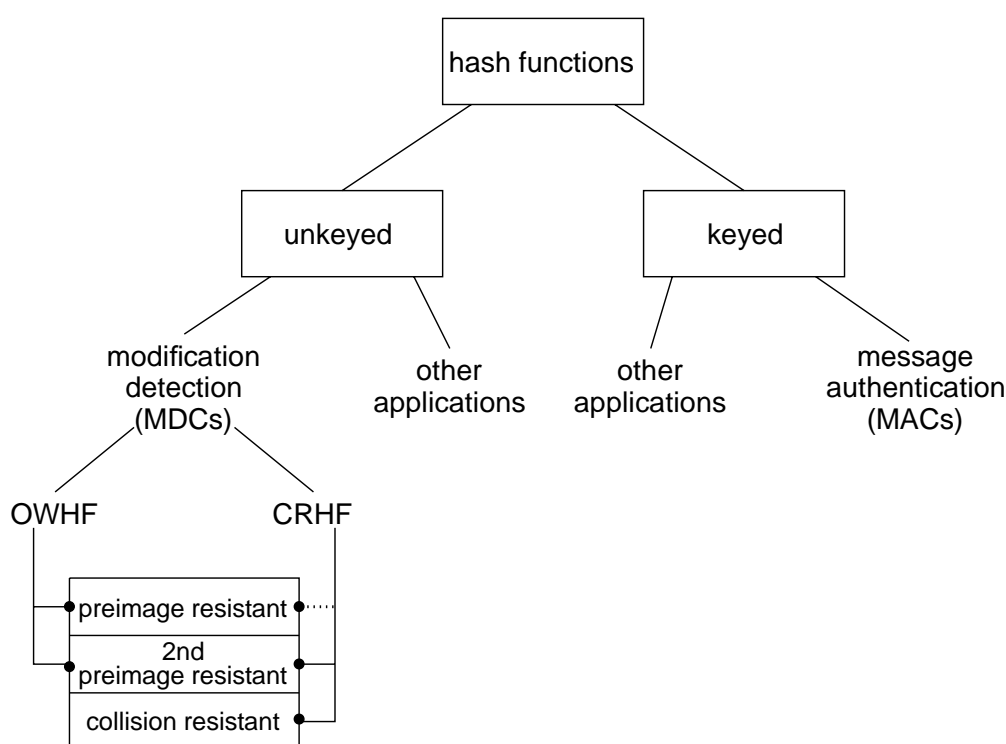


Figure 1: A simplified classification of hash functions and their applications [MvV96, Figure 9.1]. OWHF = one-way hash function. CRHF = collision-resistant hash function.

In Lecture 3 (see [knowledge base entry](#)), we came across the random oracle model for hash functions, the model in which RSA-OAEP is CCA-secure according to Fujisaki et al. [FOPS04]; we now take the opportunity of this lecture to provide further elaboration of the model.

## 2.1 Random oracle model

There are known cryptographic constructions, especially public-key constructions, based on hash functions that cannot be proven secure based only on the preceding security notions, including collision resistance.

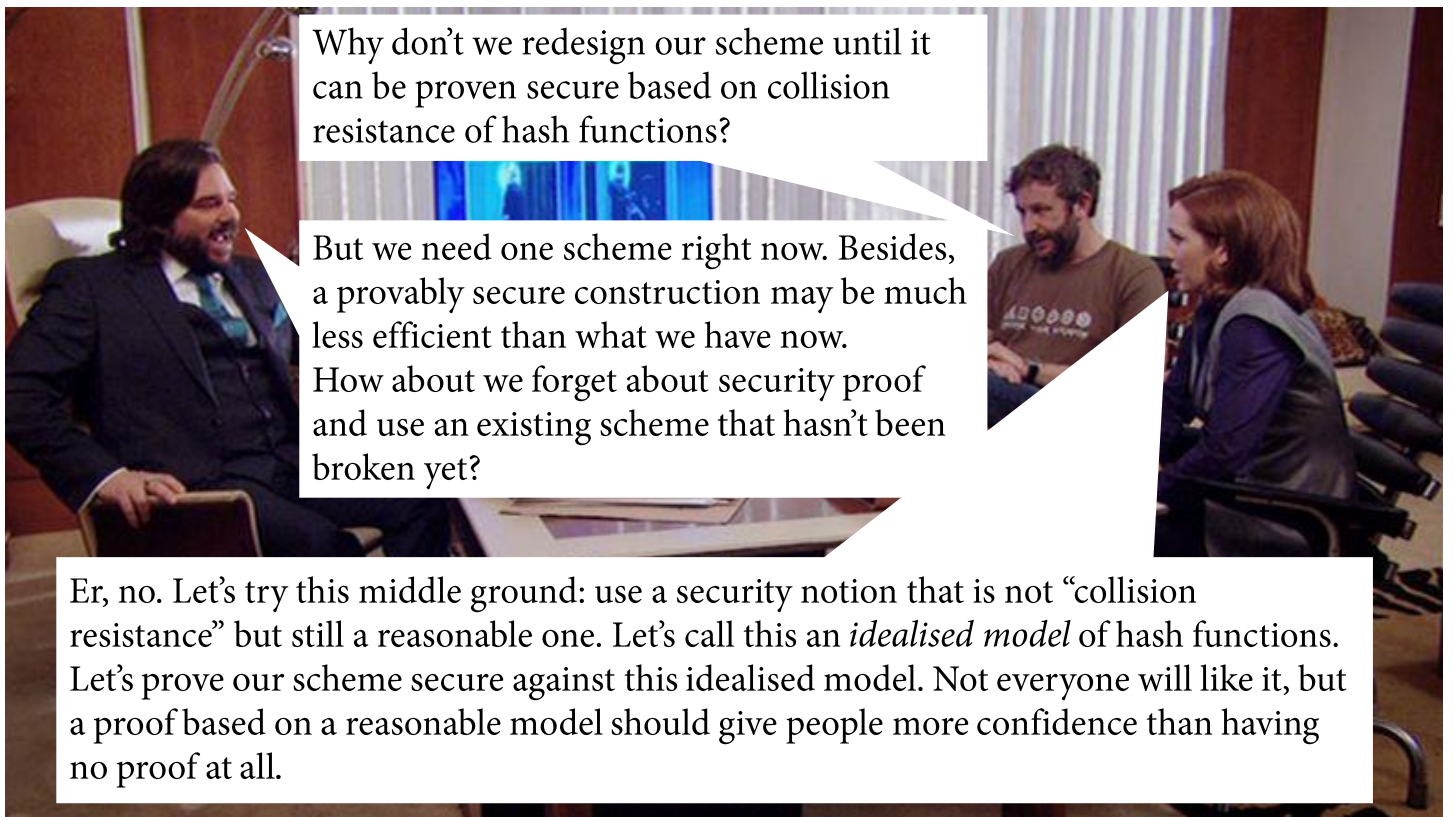


Figure 2: Justification of an idealised model.

Following a similar justification to [Figure 2](#), Bellare and Rogaway [BR93] proposed the *random oracle model*, which provides a formal methodology that can be used to design and validate cryptographic schemes in two steps [KL21, p. 188]:

1. A scheme is designed and proven secure in the random oracle model.
2. To implement the scheme in practice, a random oracle is instantiated with an "appropriately chosen" (inventors' own words) hash function.

In this model,

- All parties have access to a public but black-box, random function  $H$  that can be evaluated only by querying an oracle.
- Queries to the oracle are private so that if some party queries the oracle on input  $x$ , then no one else learns of  $x$  or that this party queries the oracle at all. Think of this as a local evaluation of a hash function.

However, if an adversary queries the oracle with  $x$ , the reduction (in a reduction-based proof) can see this query and learn  $x$ .

This property 🙌 is called *extractability*, and does not contradict the requirement that queries to the oracle are private, since the reduction is simulating the oracle.

- Random oracles are *consistent*, i.e., if an oracle ever output  $y$  for input  $x$ , then it will always output  $y$  whenever given input  $x$ .

Thus, we can define  $H$  in terms of its input-output pairs.

No party knows the algorithm of  $H$ , only its output values.

If a random oracle has not been queried with  $x$  before, then  $H(x)$  is a uniform output value.

- The random oracle model has the property of *programmability*:

In response to query  $x$ , the reduction can set the value of  $H(x)$  to a value of its choice, provided the value is uniformly distributed.

Extractability and programmability 🙌 are convenient for security proofs, but neither can be implemented by any concrete instantiation of the random oracle.

The random oracle model is widely used for proving the security of public-key schemes, but

- No concrete instantiation of a random oracle can behave like a random function since the instantiation must be fixed and its code must be known.

Instantiating the function with an off-the-shelf algorithm like SHA-256 may work for some schemes but not all.

- Thus there is no formal guarantee that security in the random oracle model carries over to security in the real world.
- In fact, there are contrived schemes that can be proven secure in the random oracle model but are insecure regardless of how the random oracle is instantiated in the second step.

Nevertheless, the random oracle model is here to stay, because

- Cryptographic schemes designed in the model are more efficient than those designed without the model.
- A security proof in the model is sound in the sense that the only possible attacks on a real-world instantiation of the scheme are those that exploit a weakness of the hash function.

The scheme can be fixed by replacing the hash function with a better one.

- There have been no successful real-world attacks on schemes proven secure in the model, when the random oracle was instantiated properly.

On the contrary, some attempts to remove random oracles from security proofs have led to modified schemes with vulnerabilities that were not present in the original schemes that were secure in the model [KM15].

### 3 Merkle-Damgård construction

🎯 Goal: To produce a fixed-length hash for an input of any length.

💡 Idea: **1** Start with a primitive that produces a fixed-length hash for a fixed-length (but longer) input. **2** Extend the primitive to cater for inputs of any length. This approach is called *domain extension*.

The screenshot shows a search engine interface with the query "random oracle model". It displays two search results. The first result is titled "[HTML] The random oracle model: a twenty-year retrospective" by N. Koblitz and A. J. Menezes, published in "Designs, Codes and Cryptography" in 2015 by Springer. The snippet mentions "words for our defense of the random oracle model, which he likened to worship of the Bronze Serpent: Indeed, what happened with the Random Oracle Model reminds us of the biblical ...". The second result is titled "The random oracle methodology, revisited" by R. Canetti, O. Goldreich, and S. Halevi, published in the "Journal of the ACM (JACM)" in 2004. The snippet states "of their oracle may be secure in the Random Oracle Model, and yet be insecure when ... in the Random Oracle Model, but for which any implementation yields insecure schemes. ...".

The Merkle-Damgård transform implements domain extension and is a building block of Message Digest 5 (MD5) and the Secure Hash Algorithms (SHA) families of hash functions:

**Definition 3: Merkle-Damgård transform [KL21, Construction 6.3]**

Let  $h$  be a *compression function* (see Definition 1) for inputs of length  $n + n' \geq 2n$  with output length  $n$ .

$n'$  is called the word size.

Fix  $\ell \leq n'$  and  $IV \in \{0, 1\}^n$ .

Construct hash function  $(\text{Gen}, H)$ :

- $\text{Gen}$  takes as input a security parameter  $1^n$  and outputs a key  $s$ , as per Definition 1.
- $H$  takes as input **1** a key  $s$  and **2** a string  $x \in \{0, 1\}^*$  of length less than  $2^\ell$  (i.e.,  $|x| < 2^\ell$ ), and do:

1. Append a 1 to  $x$ , followed by enough zeros so that the length of the resulting string is  $\ell$  is less than a multiple of  $n'$ , i.e.,

$$|x||1||0||0|| \cdots ||0| = Bn' - \ell, \quad B \in \mathbb{Z}^+.$$

2. Encode  $|x|$  as an  $\ell$ -bit string and append it to  $x||1||0||0|| \cdots ||0$ , so the resulting string is exactly  $Bn'$  bits long.
3. Parse the resulting string into the sequence of  $B$   $n'$ -bit blocks:  $x_1, \dots, x_B$ .
4. Set  $z_0 \leftarrow IV$ .
5. Compute iteratively/recursively  $z_i = h^s(z_{i-1}||x_i)$ , for  $i = 1, \dots, B$ . Thus,

$$z_B = h^s(\cdots (h^s(h^s(IV ||x_1)||x_2)|| \cdots ||x_B)),$$

as illustrated in Figure 3.  $z_i$  is sometimes called the *chaining variable* [KR11, Sec. 4.6.1.1].

6. Output  $z_B$ .

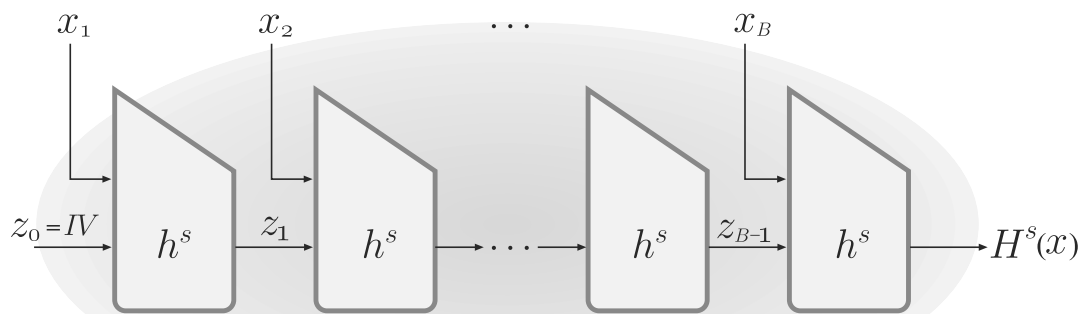


Figure 3: The Merkle-Damgård transform as defined in Definition 3. Diagram from [KL21, Figure 6.1].

**Theorem 1: [KL21, Theorem 6.4]**

$(\text{Gen}, h)$  is collision-resistant  $\implies (\text{Gen}, H)$  is collision-resistant.

*Proof.* This proof aims for the equivalent statement:  $(\text{Gen}, H)$  is *not* collision-resistant

$\implies$  (Gen,  $h$ ) is *not* collision-resistant.

Suppose we have a collision:  $x \neq x'$  but  $H^s(x) = H^s(x')$ , and:

$L = |x|$ .

$L' = |x'|$ .

The  $B$  blocks of padded  $x$  are  $x_1, \dots, x_B$ .

The  $B'$  blocks of padded  $x'$  are  $x'_1, \dots, x'_{B'}$ .

The intermediate results of  $H^s(x)$  are  $z_0, \dots, z_B$ .

The intermediate results of  $H^s(x')$  are  $z'_0, \dots, z'_{B'}$ .

Two cases to consider:

**Case  $L \neq L'$ :**  $L \neq L' \implies x_B \neq x'_{B'}$ , since as per Definition 3, the last  $\ell$  bits of  $x_B$  encodes  $L$  and similarly for  $x'_{B'}$ .

$$H^s(x) = H^s(x') \implies h^s(z_{B-1} \| x_B) = h^s(z'_{B'-1} \| x'_{B'}),$$

which is a collision with respect to  $h^s$  since  $z_{B-1} \| x_B \neq z'_{B'-1} \| x'_{B'}$ .

**Case  $L = L'$ :**  $L = L' \implies B = B'$ .

Suppose  $\lambda$  is the binary encoding of  $L$  (and  $L'$ ), then

- Padded  $x$  takes the form  $x \| 1 \| 0 \| \dots \| 0 \| \lambda$ .
- Padded  $x'$  takes the form  $x' \| 1 \| 0 \| \dots \| 0 \| \lambda$ .

$\therefore$  The last (say)  $N$  number of blocks of  $x_1, \dots, x_B$  and  $x'_1, \dots, x'_{B'}$  are the same but the rest are different, i.e.,

$$\begin{cases} x_1 \neq x'_1, \\ \vdots \\ x_{B-N} \neq x'_{B-N}, \end{cases} \quad \begin{cases} x_{B-N+1} = x'_{B-N+1}, \\ \vdots \\ x_B = x'_B, \end{cases}$$

Going back to collision:

$$\begin{aligned} H^s(x) = H^s(x') &\implies h^s(z_{B-1} \| x_B) = h^s(z'_{B-1} \| x_B) \\ &\implies h^s(h^s(z_{B-2} \| x_{B-1}) \| x_B) = h^s(h^s(z'_{B-2} \| x_{B-1}) \| x_B) \\ &\implies \dots \end{aligned}$$

$$\begin{aligned} h^s(\dots h^s(z_{B-N-1} \| x_{B-N}) \| \dots \| x_{B-1}) \| x_B &= h^s(\dots h^s(z'_{B-N-1} \| x'_{B-N}) \| \dots \| x_{B-1}) \| x_B \\ \implies h^s(z_{B-N-1} \| x_{B-N}) &= h^s(z'_{B-N-1} \| x'_{B-N}), \end{aligned}$$

which is a collision with respect to  $h^s$  since  $z_{B-N-1} \| x_{B-N} \neq z'_{B-N-1} \| x'_{B-N}$ .

□

The question now becomes how we define the compression function  $h$ . Suppose we use a block cipher as a building block and define

$$h^s(z_{i-1}, x_i) = \text{Enc}_a(b) \oplus c, \quad (1)$$

where Enc is the encryption function of block cipher.

- Suppose each of  $a$ ,  $b$  and  $c$  can be one of **1**  $s$ , **2**  $z_{i-1}$ , **3**  $x_i$ , and **4**  $z_{i-1} \| x_i$ . This translates to  $4^3 = 64$  possible combinations [KR11, Sec. 4.6.1.2].
- 20 of the combinations are collision-resistant in the black-box model (in which the block cipher appears as an unexploitable black box to any adversary) [BRS02] but the 3 of them shown in Figure 4 are particularly important.

- As [Figure 4](#) shows, the Matyas-Meyer-Oseas and Miyaguchi-Preneel constructions suffer from the constraint that the key size and block size must be the same, whereas the Davies-Meyer construction does not.

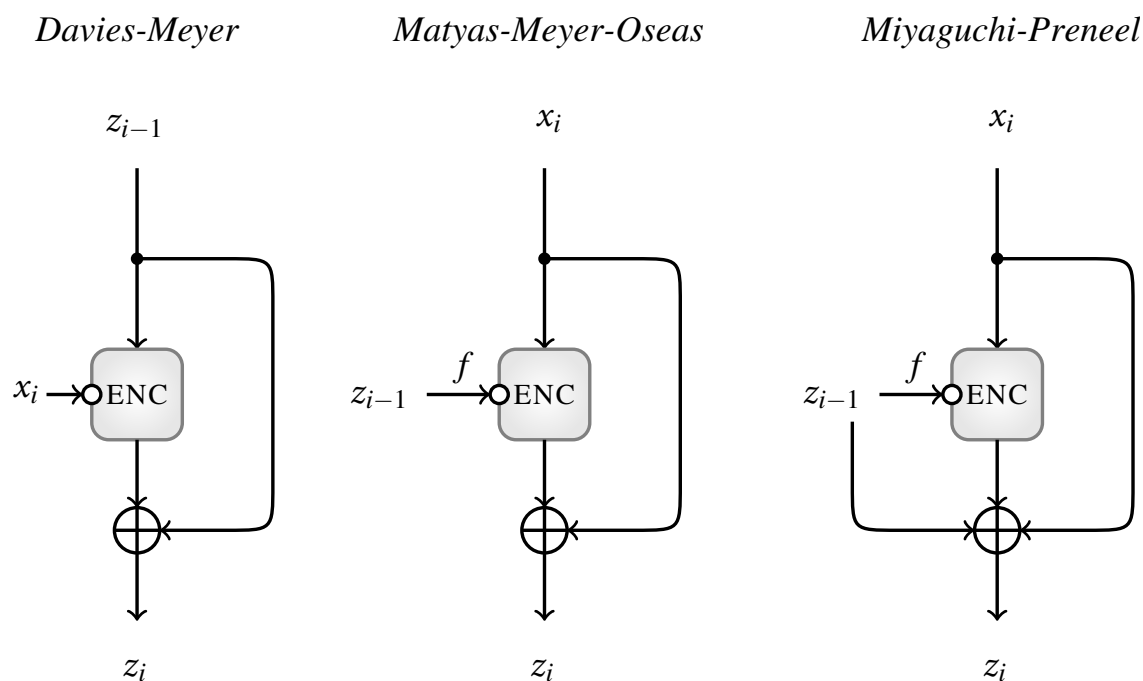


Figure 4: Three most important constructions of the compression function in Definition 3. Diagram adapted from [KR11, Fig. 4.12].

- Referring to Eq. (1), the Davies-Meyer construction equates  $a = x_i$ ,  $b = z_{i-1}$  and  $c = z_{i-1} \parallel x_i$ . **⚠** Not equating  $b = z_{i-1} \parallel x_i$  incurs the need to revise the proof for Theorem 1.
- The Davies-Meyer construction can serve as a general mechanism for converting a pseudorandom permutation to a pseudorandom function [Jou09, p. 241].
- Key security property of the Davies-Meyer construction: in the black-box model, if key length  $\geq$  block length  $n$ , then finding a second preimage requires approximately  $2^n$  encryptions, while finding a collision requires approximately  $2^{n/2}$  encryptions [vJ11, p. 312]. **👉** This suggests a desired block length of  $n = 256$  bits.
- 256-bit block ciphers are not standard, so the Davies-Meyer construction is rarely used to construct collision-resistant hash functions from block ciphers. It has nevertheless been applied to the construction of MD5 and SHA-2 from an internal block cipher structure [vJ11, pp. 312-313].

☠️ MD5 is broken and must not be used.

FIPS 180-4 [NIS15] specifies SHA-1 and SHA-2 families of hash functions.

- SHA-1 is not collision-resistant and should not be used, so we shall focus on SHA-2.

### 3.1 SHA-2

SHA-2 is the family of algorithms consisting of SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224 and SHA-512/256; see [Table 1](#).

Table 1: Members of SHA-1 and SHA-2 specified in [NIS15]. Note **1** SHA-224 and SHA-384 are truncated versions of SHA-256 and SHA-512 respectively, computed with different initial values; **2** SHA-512/224 and SHA-512/256 are truncated versions of SHA-512; **3** word size ( $\neq n'$  in Definition 3) can be 32 or 64 bits depending on the algorithm.

Algorithm	Message Size (bits)	Block Size (bits)	Word Size (bits)	Message Digest Size (bits)
SHA-1	$< 2^{64}$	512	32	160
SHA-224	$< 2^{64}$	512	32	224
SHA-256	$< 2^{64}$	512	32	256
SHA-384	$< 2^{128}$	1024	64	384
SHA-512	$< 2^{128}$	1024	64	512
SHA-512/224	$< 2^{128}$	1024	64	224
SHA-512/256	$< 2^{128}$	1024	64	256

Each algorithm can be described in two stages [NIS15], consistent with the Merkle-Damgård construction:

- 1. Preprocessing:** This involves padding a message, parsing the padded message into fixed-length blocks, and setting IVs to be used in the hash computation.
- 2. Hash computation:** This generates a message schedule from the padded message and uses that schedule, along with functions, constants, and word operations to *iteratively* generate a series of hash values. The final hash value generated by the hash computation is used to determine the message digest.

In the following, we focus on SHA-256.

- SHA-256 sets the hash size  $n = 256$ , the block size  $n' = 512$  and  $\ell = 64$  in Definition 3.
- Like how we approached the AES, we shall study the algorithm in conjunction with the [OpenSSL code](#) distributed in:
  1. `crypto/sha/sha256.c`
  2. `crypto/evp/e_aes_cbc_hmac_sha256.c`
  3. `include/crypto/md32_common.h`

### 3.1.1 Preprocessing

Preprocessing consists of three steps following the Merkle-Damgård transform in Definition 3:

1. Padding the message [NIS15, Sec. 5.1.1]:
  - » Append a 1 and enough number of zeros to the message so that the length of the resulting string is 64 bits less than a multiple of 512 bits.
  - » If the message length is  $L$  bits, then the number of zero bits can be obtained by calculating:

$$512k - L - 1 - 64 \bmod 512 \equiv -L - 1 - 64 \bmod 512,$$

where  $k \in \mathbb{N}$ .

» OpenSSL code omitted.

### Quiz 1

Suppose we want to hash “COMP5074”, which in ASCII encoding occupies  $8 \times 8 = 64$  bits. How many zero bits do we need for the padding?

2. Parsing the message into message blocks [NIS15, Sec. 5.2.1]:

- » The padded message is parsed into 512-bit blocks.
- » Each block consists of 16 32-bit words.
- » OpenSSL code omitted.

3. Setting the initial hash value [NIS15, Sec. 5.3.3]:

- » This is the same as IV in Definition 3 and it consists of 8 32-bit words.
- » This is initialised to the first 32 bits of the fractional parts of the square roots of the first 8 prime numbers.
- » These 8 words can be generated using the following Python code:

```
from math import sqrt
[hex(int((sqrt(i) - int(sqrt(i)))*2**32)) for i in (2,3,5,7,11,13,17,19)]
```

- » See Listing 1.

Listing 1: SHA256\_Init.

```
int SHA256_Init(SHA256_CTX *c)
{
    memset(c, 0, sizeof(*c));
    c->h[0] = 0x6a09e667UL;
    c->h[1] = 0xbb67ae85UL;
    c->h[2] = 0x3c6ef372UL;
    c->h[3] = 0xa54ff53aUL;
    c->h[4] = 0x510e527fUL;
    c->h[5] = 0x9b05688cUL;
    c->h[6] = 0x1f83d9abUL;
    c->h[7] = 0x5be0cd19UL;
    c->md_len = SHA256_DIGEST_LENGTH;
    return 1;
}
```

### 3.1.2 Hash computation

At the end of preprocessing,

- The padded message would have been parsed into  $B$  512-bit blocks:

$$M^{(1)}, M^{(2)}, \dots, M^{(B)}.$$

Notation 🙌 made consistent with [NIS15]. Each block consists of 16 32-bit words.

- Initial hash value is stored in  $H^{(0)}$ .
- Based on [NIS15, Sec. 6.2.2], hash computation iterates through each block as follows.

For  $i = 1$  to  $B$  (iterating through the message blocks):

**Step 1:** For each block, prepare the message schedule consisting of 64 32-bit words denoted by  $W_0, \dots, W_{63}$ :

$$W_t = \begin{cases} M_t^{(i)} & 0 \leq t \leq 15, \\ \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16} & 16 \leq t \leq 63, \end{cases}$$

where functions  $\sigma_0$  and  $\sigma_1$  are defined as:

$$\begin{aligned} \sigma_0(W) &= (W \ggg 7) \oplus (W \ggg 18) \oplus (W \gg 3), \\ \sigma_1(W) &= (W \ggg 17) \oplus (W \ggg 19) \oplus (W \gg 10). \end{aligned}$$

Above, the operator  $\ggg$  right-rotates and the operator  $\gg$  right-shifts.

Think of  $M^{(i)}$  as a key; and  $W_0, \dots, W_{63}$  as the expanded subkeys to be fed to instantiations of the Davies-Meyer construction (see [Figure 4](#)) in Step 3 later.

**Step 2:** Initialise the eight working variables:

$$a = H_0^{(i-1)}, \quad b = H_1^{(i-1)}, \quad \dots, \quad h = H_7^{(i-1)}.$$

**Step 3:** For  $t = 0$  to 63 (iterating through the 64 words of a message block):

$T_1 = h + \Sigma_1(e) + \text{Ch}(e, f, g) + K_t + W_t,$	$\Sigma_0$ and $\Sigma_1$ are GF(2)-linear functions [ <a href="#">NIS15</a> , Sec. 4.1.2]:
$T_2 = \Sigma_0(a) + \text{Maj}(a, b, c),$	$\Sigma_0(a) = a \ggg 2 \oplus a \ggg 13 \oplus a \ggg 22,$
$h = g,$	$\Sigma_1(e) = e \ggg 6 \oplus e \ggg 11 \oplus e \ggg 25.$
$g = f,$	$\text{Ch}$ and $\text{Maj}$ are boolean functions [ <a href="#">NIS15</a> , Sec. 4.1.2]:
$f = e,$	$\text{Ch}(e, f, g) = (e \wedge f) \oplus (\neg e \wedge g),$
$e = d + T_1,$	$\text{Maj}(a, b, c) = (a \wedge b) \oplus (a \wedge c) \oplus (b \wedge c).$
$d = c,$	Constants $K_0, \dots, K_{63}$ are the first 32 bits of the fractional parts of the cube roots of the first 64 prime numbers [ <a href="#">NIS15</a> , Sec. 4.2.2]; see <a href="#">Listing 2</a> .
$c = b,$	
$b = a,$	
$a = T_1 + T_2.$	

The preceding operations implement the *encryption part* of the Davies-Meyer construction,  $\text{Enc}_{W_t}(H_0^{(i-1)} \parallel \dots \parallel H_7^{(i-1)})$ ; and are visualised in [Figure 5](#).

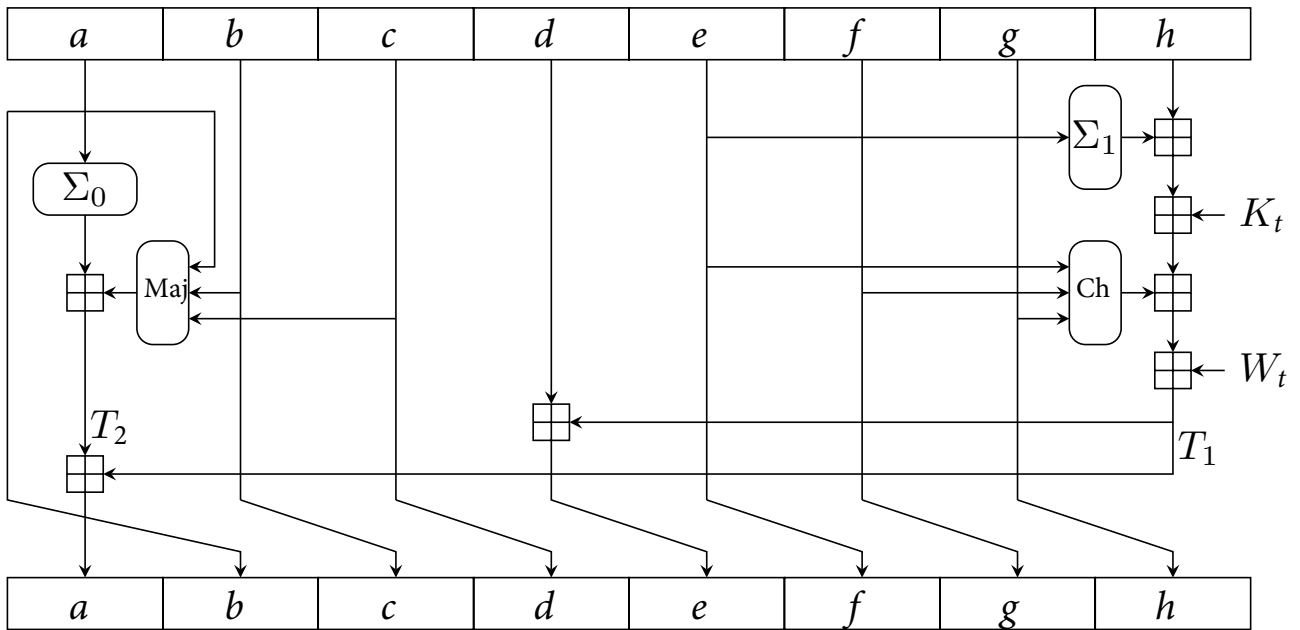


Figure 5: The encryption part of the Davies-Meyer construction,  $\text{Enc}_{W_t}(H_0^{(i-1)} \parallel \dots \parallel H_7^{(i-1)})$ , of SHA-256. Diagram adapted from [MNS11, Fig. 1].

**Step 4:** This step and the preceding complete the Davies-Meyer operation:

$$H_0^{(i)} = a + H_0^{(i-1)}, \quad H_1^{(i)} = b + H_1^{(i-1)}, \quad \dots, \quad H_7^{(i)} = h + H_7^{(i-1)},$$

where  $+$  is modulo  $2^{32}$ .

The final message digest is  $H_0^{(B)} \parallel H_1^{(B)} \parallel \dots \parallel H_7^{(B)}$ .

A clear match can be made between the operations described above and Listing 2.

Listing 2: sha256\_block\_data\_order.

```
static const SHA_LONG K256[64] = {
    0x428a2f98UL, 0x71374491UL, 0xb5c0fbcfUL, 0xe9b5dba5UL,
    0x3956c25bUL, 0x59f111f1UL, 0x923f82a4UL, 0xab1c5ed5UL,
    0xd807aa98UL, 0x12835b01UL, 0x243185beUL, 0x550c7dc3UL,
    0x72be5d74UL, 0x80deb1feUL, 0x9bdc06a7UL, 0xc19bf174UL,
    0xe49b69c1UL, 0xefbe4786UL, 0x0fc19dc6UL, 0x240ca1ccUL,
    0x2de92c6fUL, 0x4a7484aaUL, 0x5cb0a9dcUL, 0x76f988daUL,
    0x983e5152UL, 0xa831c66dUL, 0xb00327c8UL, 0xbf597fc7UL,
    0xc6e00bf3UL, 0xd5a79147UL, 0x06ca6351UL, 0x14292967UL,
    0x27b70a85UL, 0x2e1b2138UL, 0x4d2c6dfcUL, 0x53380d13UL,
    0x650a7354UL, 0x766a0abbUL, 0x81c2c92eUL, 0x92722c85UL,
    0xa2bfe8a1UL, 0xa81a664bUL, 0xc24b8b70UL, 0xc76c51a3UL,
    0xd192e819UL, 0xd6990624UL, 0xf40e3585UL, 0x106aa070UL,
    0x19a4c116UL, 0x1e376c08UL, 0x2748774cUL, 0x34b0bcb5UL,
    0x391c0cb3UL, 0x4ed8aa4aUL, 0x5b9cca4fUL, 0x682e6ff3UL,
    0x748f82eeUL, 0x78a5636fUL, 0x84c87814UL, 0x8cc70208UL,
    0x90beffffaUL, 0xa4506cebUL, 0xbef9a3f7UL, 0xc67178f2UL
};

static void sha256_block_data_order(SHA256_CTX *ctx, const void *in, size_t num)
{
    unsigned MD32_REG_T a, b, c, d, e, f, g, h, s0, s1, T1, T2;
    SHA_LONG X[16], l;
    int i;
    const unsigned char *data = in;

    while (num--) {
        a = ctx->h[0]; b = ctx->h[1]; c = ctx->h[2]; d = ctx->h[3];
```

```

e = ctx→h[4]; f = ctx→h[5]; g = ctx→h[6]; h = ctx→h[7];

for (i = 0; i < 16; i++) { /* W_0, ..., W_15 */
    (void)HOST_c2l(data, l);
    T1 = X[i] = l;
    T1 += h + Sigma1(e) + Ch(e, f, g) + K256[i];
    T2 = Sigma0(a) + Maj(a, b, c);
    h = g;
    g = f;
    f = e;
    e = d + T1;
    d = c;
    c = b;
    b = a;
    a = T1 + T2;
}

for (; i < 64; i++) { /* W_16, ..., W_63 */
    s0 = X[(i + 1) & 0x0f];
    s0 = sigma0(s0);
    s1 = X[(i + 14) & 0x0f];
    s1 = sigma1(s1);

    T1 = X[i & 0xf] += s0 + s1 + X[(i + 9) & 0xf];
    T1 += h + Sigma1(e) + Ch(e, f, g) + K256[i];
    T2 = Sigma0(a) + Maj(a, b, c);
    h = g;
    g = f;
    f = e;
    e = d + T1;
    d = c;
    c = b;
    b = a;
    a = T1 + T2;
}

ctx→h[0] += a; ctx→h[1] += b; ctx→h[2] += c; ctx→h[3] += d;
ctx→h[4] += e; ctx→h[5] += f; ctx→h[6] += g; ctx→h[7] += h;
}
}

```

## 4 Sponge construction

Whereas the Merkle-Damgård construction is compression-based, the sponge construction is a permutation-based method for constructing hash functions.

Sponge construction = *iterated* construction for building a function  $F$  with a *variable-length input* and a *variable-length output* based on a *fixed-length permutation*  $f$  operating on a fixed number  $b$  of bits ( $b$  is called the *width*) [BDPVA11a].

- $f$  can also be a transformation instead of a permutation, but for our focus here, we shall restrict  $f$  to permutation.
- A *sponge function* instantiates the sponge construction. Equivalently, an instance of the sponge construction is a sponge function.

- Why “sponge”? An arbitrary number of input bits are “absorbed” into the state of the function, after which an arbitrary number of output bits are “squeezed” out of its state [Dwo15, Sec. 4].

More precisely,

#### Definition 4: Sponge construction [BDPVA11a, Sec. 2.2]

The sponge construction builds a function  $\text{SPONGE}[f, \text{pad}, r]$  with domain  $\{0, 1\}^*$  and co-domain  $\{0, 1\}^\infty$  using **1** a fixed-length permutation  $f$ , **2** a sponge-compliant padding rule  $\text{pad}$ , and **3** a bitrate  $r$ .

- A padding rule is sponge-compliant if it never results in the empty string and if it satisfies the criterion:

$$\forall n \geq 0, \forall M, M' \in \{0, 1\}^* : M \neq M' \implies M \parallel \text{pad}(|M|) \neq M' \parallel \text{pad}(|M'|) \parallel 0^{nr}.$$

As a sufficient condition, a padding rule that is reversible (i.e., injective), non-empty and such that the last block must be non-zero, is sponge-compliant [BDPVA11a, Sec. 2.2].

- $f$  operates on  $b$  bits;  $b$  is called the *width* of  $f$ .
- Bitrate  $r$  is the width of each message block to be processed later.

Initialisation (see Figure 6):

- The input message is padded and parsed into  $r$ -bit blocks.
- The sponge construction has a state of  $b = r + c$  bits, all of which initialised to zero, where  $c$  is called the *capacity*.

The first  $r$  bits of the state are called the *outer state*.

The last  $c$  bits of the state are called the *inner state*.

- In subsequent processing, the outer state and inner state are processed differently.

Computation happens in two phases (see Figure 6):

- Absorbing phase:** The  $r$ -bit input message blocks are XORed into the outer part of the state, interleaved with applications of the function  $f$ .  
When all message blocks are processed, the sponge construction switches to the squeezing phase.
- Squeezing phase:** The outer part of the state is iteratively returned as output blocks, interleaved with applications of  $f$ .  
The number of iterations is determined by the requested number of bits  $\ell$ .

Finally the output is truncated to its first  $\ell$  bits.

 Note:

- **i** Co-domain  $\neq$  range; co-domain is the set of potential output values of a function, whereas range is the set of actual output values of a function.
- The  $c$ -bit inner state is never directly affected by the input blocks and never output during the squeezing phase.
- The capacity  $c$  determines the attainable security level of the construction.

- A sponge function is random if  $f$  is random.

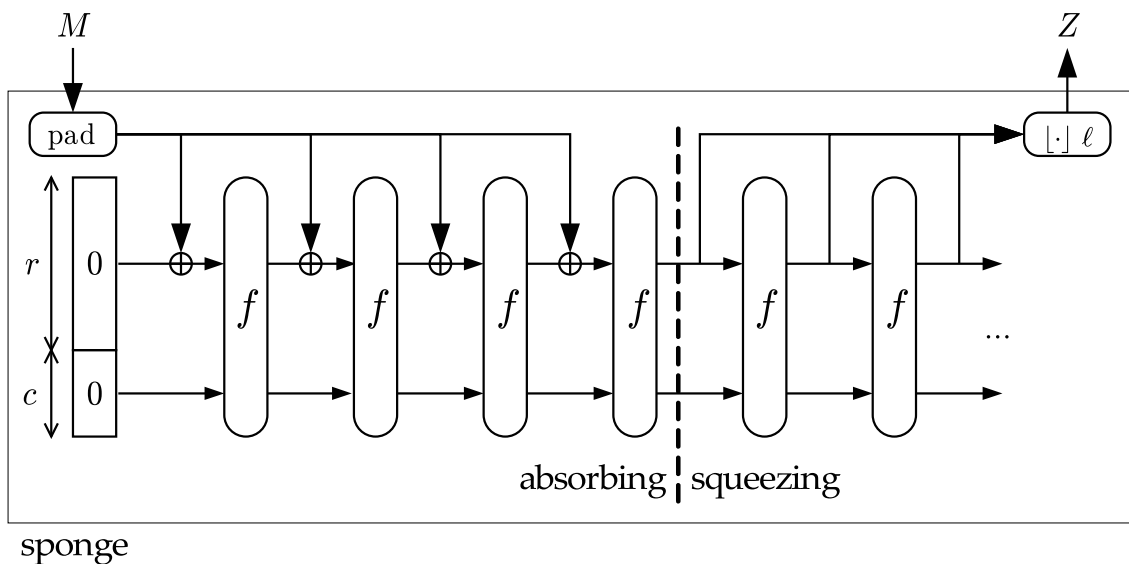


Figure 6: Sponge function [BDPVA11a, Figure 2.1].

### Example 1

Here is one way to instantiate a hash function from the sponge construction [KL21, Construction 7.6].

Fix the number of squeezing stages as  $\lambda$ , and the number of output bits from each squeezing stage as  $\nu$ , as shown in Figure 7; so the final message digest is  $\lambda\nu$  bits long.

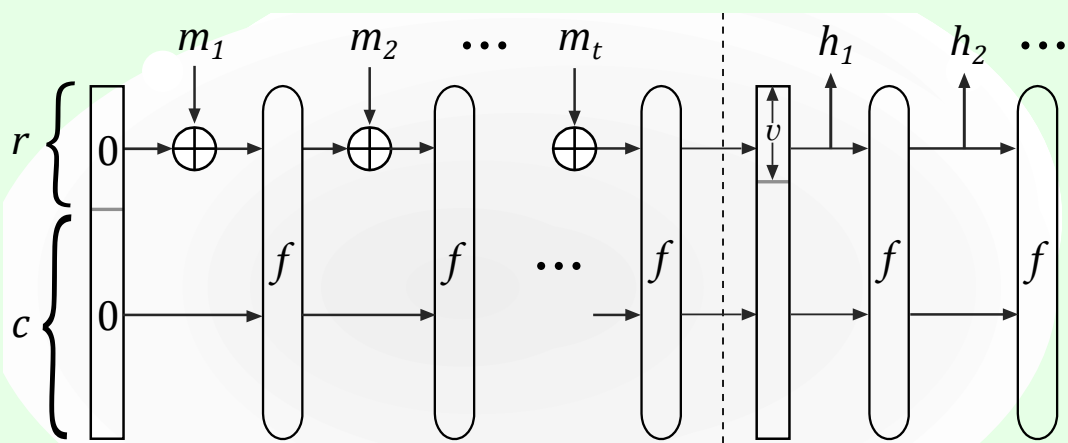


Figure 7: Instantiating a hash function from the sponge construction [KL21, Figure 7.12].

Initialisation:

- Apply the same padding rule as that for the Merkle-Damgård construction in Definition 3.
- Parse the padded message into  $m_1, \dots, m_t$ .
- Initialise the state  $y$  to zero, i.e.,  $y_0 = 0^b$ .

Absorbing phase: For  $i = 1, \dots, t$ , update the state as

$$y_i = f(y_{i-1} \oplus (m_i \| 0^c)).$$

Squeezing phase:

- Set  $y_1^* = y_t$  and  $h_1 =$  the first  $v$  bits of  $y_1^*$ .
- For  $i = 2, \dots, \lambda$ ,

$$y_i^* = f(y_{i-1}^*), \quad h_i = \text{first } v \text{ bits of } y_i^*.$$

Output:  $h_1 \parallel \dots \parallel h_\lambda$ .

Design motivation/rationale for the construction:

- So that it can be used to specify a function that behaves like a random oracle, with the sole exception that it would have inner (not outer) collisions.  
A random sponge function is by design/definition as strong as a random oracle, except for the effects induced by finite memory.
- Why random oracle? So that this model can be used as an alternative to the random oracle model for expressing security claims, i.e., the sponge function was designed for designers to be able to formulate compact security claims [BDPVA07].
- Sponge construction and its sister construction — the *duplex construction* [BDPVA12] — can be used to design cryptographic schemes other than hash functions, e.g., stream ciphers, block ciphers, message authentication codes and authenticated encryption schemes.

#### Definition 5: Duplex construction [BDPVA11a, Sec. 2.3]

The duplex construction builds a function  $\text{DUPLEX}[f, \text{pad}, r]$  with domain  $\{0, 1\}^*$  and co-domain  $\{0, 1\}^\infty$  using **1** a fixed-length permutation  $f$ , **2** a sponge-compliant padding rule  $\text{pad}$ , and **3** a bitrate  $r$ .

- Sponge-compliance, width  $b$  of  $f$  and bitrate are as defined in Definition 4.
- Unlike a sponge function that is stateless in between calls, the duplex construction results in an object — called a *duplex object* — that accepts calls that take an input string and return an output string that depends on *all the inputs* received so far.

Initialisation (see Figure 8): Same as that of the sponge construction in Definition 4.

Computation takes place in the form of duplexing calls (see Figure 8):

- If  $D$  denotes a duplex object, then  $D.\text{duplexing}(\sigma, \ell)$  denotes a duplexing call, with  $\sigma$  as the input string and  $\ell$  the requested number of bits, where  $\ell \leq r$ .
- The maximum bit-length of  $\sigma$  is the *maximum duplex rate*, defined as the largest number such that the longest padded  $\sigma$  is at most  $r$  bits long.
- Upon receiving a  $D.\text{duplexing}(\sigma, \ell)$  call,  $D$  pads  $\sigma$  and XORs it into the outer state. Then it applies  $f$  to the state and returns the first  $\ell$  bits of the outer state at the output.
- The duplexing call is a *blank call* if  $\sigma$  is empty; or a *mute call* if  $\ell = 0$ .

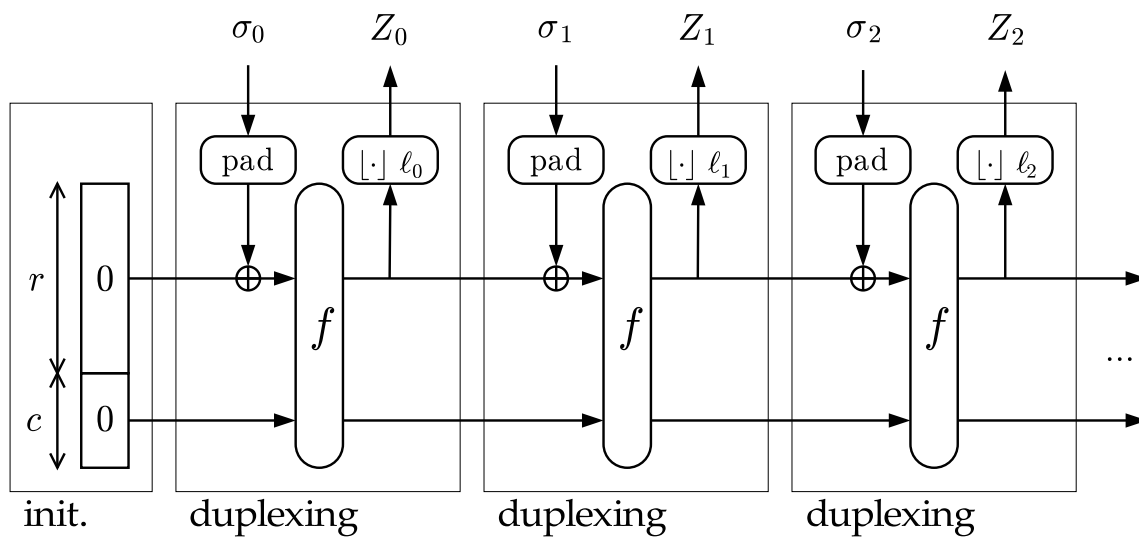


Figure 8: Duplex construction [BDPVA11a, Figure 2.2].

Why the duplex construction?

- It is the first mode of this kind to be directly based on a permutation instead of a block cipher [BDPVA12].

Permutation is more lightweight and efficient than a block cipher.

- More for authenticated encryption: It enables the alternation of input and output blocks at the same rate as the sponge construction, like a full-duplex communication [BDPVA12].

The input blocks of the duplex are used to inject the key and the message blocks, while the intermediate output blocks are used as a key stream and the last one as a MAC tag [BDPVA12].

- The output of a duplexing call can be obtained by evaluating a sponge function **1** with the same parameters, **2** on the input constructed from all the previous inputs to the duplex object (see Figure 9), so the duplex construction inherits the security properties from the sponge construction [BDPVA11a, p. 14, Sec. 6.4].

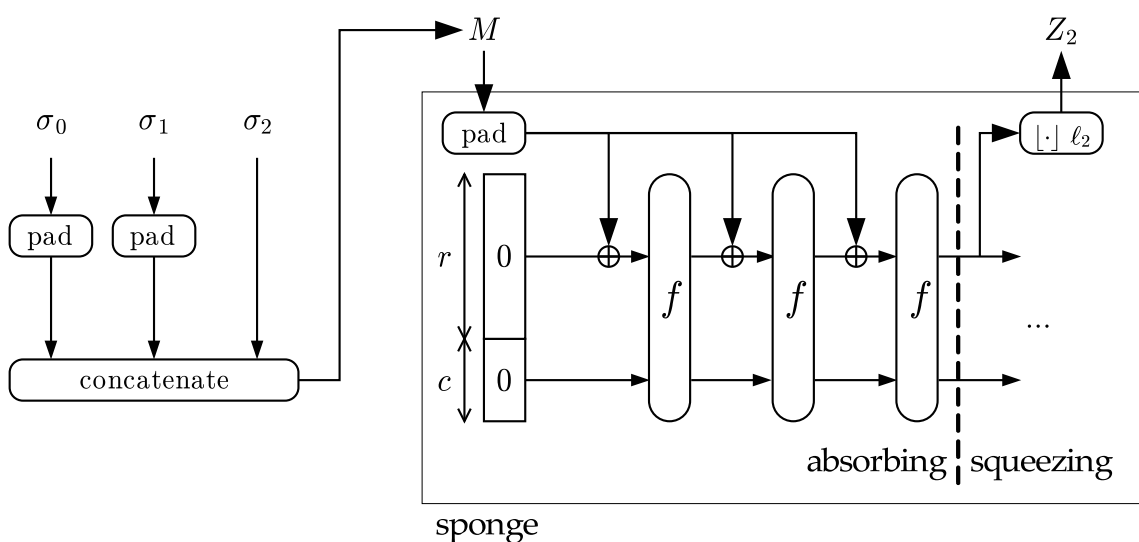


Figure 9: Generating the output of a duplexing call with a sponge function [BDPVA12, Fig. 2].

## 4.1 SHA-3

In 2007, after practical collision attacks were reported on MD5, and weaknesses were found in SHA-1, NIST announced the SHA-3 Cryptographic Hash Algorithm Competition to design a new, stronger hash function.

- SHA-3 was intended to be a backup to SHA-2 because of the long overhead to develop a new standard [Bou15].



Figure 10: Photo of the KECCAK team on [Twitter](#).

In 2012, NIST announced the KECCAK family of cryptographic primitives to be the winner.

In 2015 (less than a decade ago!), NIST released FIPS 202 [Dwo15], standardising the KECCAK-based SHA-3 family of functions.

KECCAK is the family of all *sponge functions* with a KECCAK- $f$  *permutation* (see Definition 6) as the underlying function and *multi-rate padding* (see Definition 7) as the padding rule, as originally specified in [BDPVA11b].

### Definition 6: KECCAK- $f$ and KECCAK- $p$

KECCAK- $f[b]$  is the family of 7 permutations originally specified in [BDPVA11b] as the underlying function for Keccak, where the width (as per Definition 4) of permutation  $b \in \{25, 50, 100, 200, 400, 800, 1600\}$ .

KECCAK- $[c] \equiv$  KECCAK- $f[1600]$  with capacity (as per Definition 4)  $c$ .

KECCAK- $p[b, n_r]$  is a generalisation of KECCAK- $f[b]$  defined in FIPS 202 [Dwo15] by converting the number of rounds  $n_r$  to an input parameter.

### Definition 7: Multi-rate padding [BDPVA11b, Definition 1]

Multi-rate padding, denoted by  $\text{pad}_{10^*1}$ , appends a single bit 1 followed by the minimum number of bits 0 followed by a single bit 1 such that the length of the result is a multiple of the block length.

The SHA-3 family consists of

- 4 hash functions: SHA3-224, SHA3-256, SHA3-384, SHA3-512; and
- 2 extendable-output (XOF) functions: SHAKE128, SHAKE256.

### Definition 8: Extendable-output function (XOF) [Dwo15, BDH<sup>+</sup>22]

A generalisation of a hash function that on input of a bit string outputs a digest whose length can be extended to any desired length.

Additionally, requesting more output bits costs only the production of these new bits.

All functions can be considered to be *modes of operation* of the KECCAK- $p[1600, 24]$  permutation.

The ensuing discussion of SHA-3 is divided into the state and permutation.

### 4.1.1 State

The  $\text{KECCAK-}p$  permutation is repeatedly applied to the 3D *state* depicted in [Figure 11](#), of dimensions  $5 \times 5 \times 2^\ell$  ( $\ell \in \{0, 1, \dots, 6\}$ ); the state has  $b = 5 \times 5 \times w$  ( $w = 2^\ell$ ) number of bits.

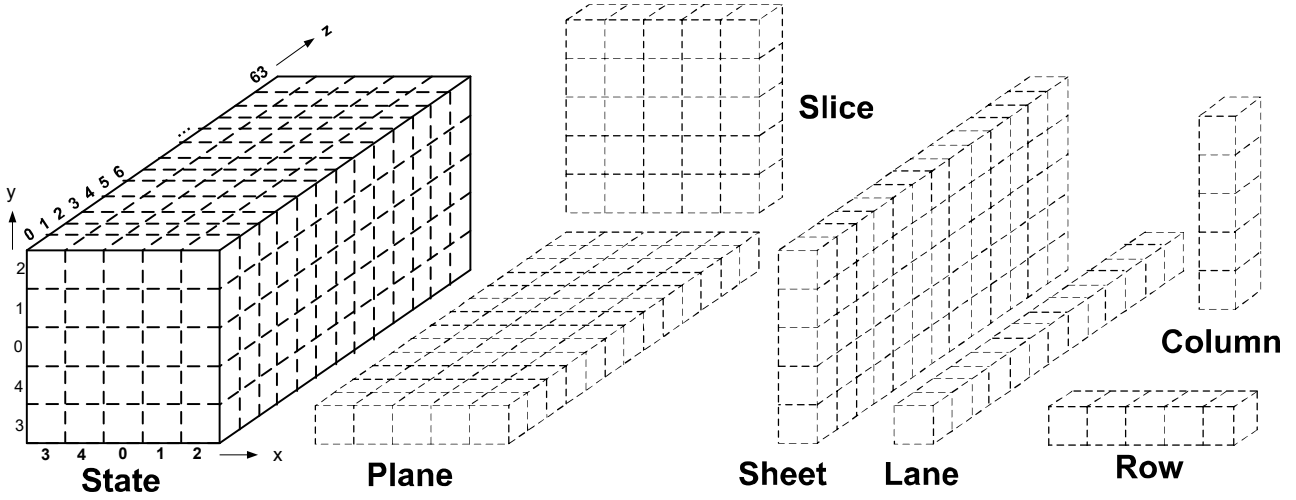


Figure 11: Parts of a sample SHA-3 state array, where  $\ell = 6$ ,  $w = 64$ ,  $b = 5 \times 5 \times w = 1600$  [LFF<sup>+</sup>15, Figure 2]. Coordinates  $x = y = 0$  mark the center of a slice.

In [Figure 11](#),

One-dimensional parts:

- A *lane* is a set of  $w$  bits for fixed  $x, y$ .
- A *row* is a set of 5 bits for fixed  $y, z$ .
- A *column* is a set of 5 bits for a fixed  $z, x$ .

Two-dimensional parts:

- A *sheet* is a set of  $5w$  bits for a fixed  $x$ .
- A *plane* is a set of  $5w$  bits for a fixed  $y$ .
- A *slice* is a set of  $25$  bits for a fixed  $z$ .

In analysis, a state bit is indexed based on the Cartesian coordinate system in [Figure 11](#), where  $0 \leq x, y \leq 4$  and  $0 \leq z \leq w - 1$ , and denoted by

$$A[x, y, z].$$


In implementation, the state is stored in a flattened form, i.e.,

$$S = S[0] \| S[1] \| \dots \| S[b - 1],$$

which is related to  $A[x, y, z]$  by

$$A[x, y, z] = S[(x + 5y)w + z], \quad (2)$$

i.e.,  $A$  is serialised into  $S$  lane-first (lane as defined in [Figure 11](#)), then by  $x$  and  $y$ .

 Expressions in the  $x$  and  $y$  coordinates should be taken modulo 5 while expressions in the  $z$  coordinate modulo  $w$ .

The state is initially set to the input values of the permutation.

### 4.1.2 Permutation

The core of SHA-3 is the  $\text{KECCAK-}p$  permutation.

A round of a  $\text{KECCAK-}p$  permutation, denoted by  $\text{Rnd}$ , consists of a sequence of five successive transformations called *step mappings* and denoted by  $\theta, \rho, \pi, \chi, \iota$ , all of which operate on the state.

**Step mapping  $\theta$**  (see Figure 12):

$$A[x, y, z] \leftarrow A[x, y, z] + \sum_{j=0}^4 A[x - 1, j, z] + \sum_{j=0}^4 A[x + 1, j, z - 1]. \quad (3)$$

$\theta$  is the *column parity mixing layer* [SD18] that adds to each bit  $A[x, y, z]$  the bitwise sum of the parities of two columns: that of  $A[x - 1, *, z]$  and that of  $A[x + 1, *, z - 1]$ .

$\theta$  is *linear* and aimed at *diffusion* and is *translation-invariant* in all directions [BDPVA11b, Sec. 2.3.2].

- Translation-invariance enables the introduction of a size parameter that can be varied without having to re-specify the step mappings [BDPVA11b, Sec. 2.2].
- Translation-invariance enables the *Matryoshka structure*, where a smaller-width structure can be reused as a symmetric structure in a larger-width structure (recall  $w = 2^\ell$ , i.e., every smaller lane length divides a larger lane length).

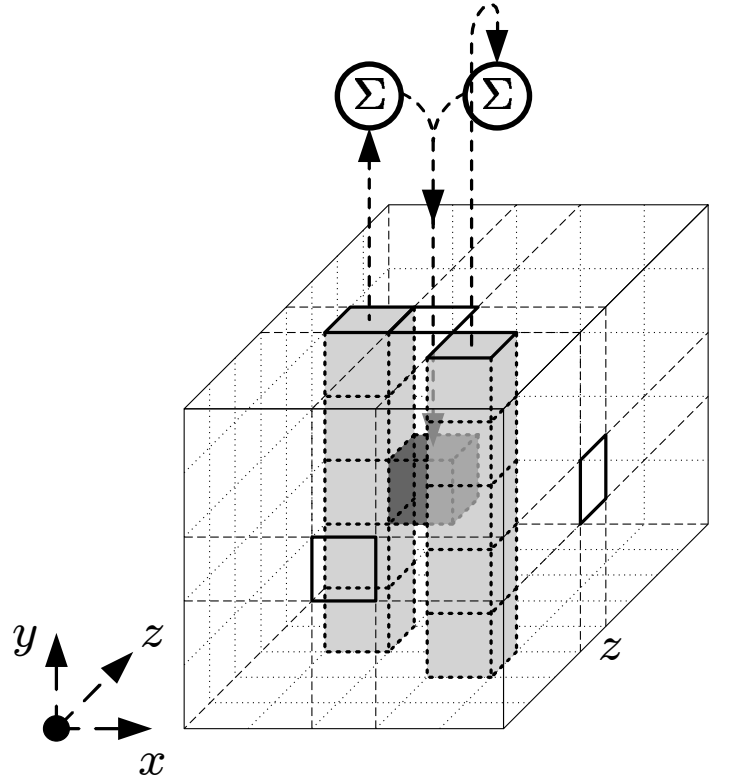


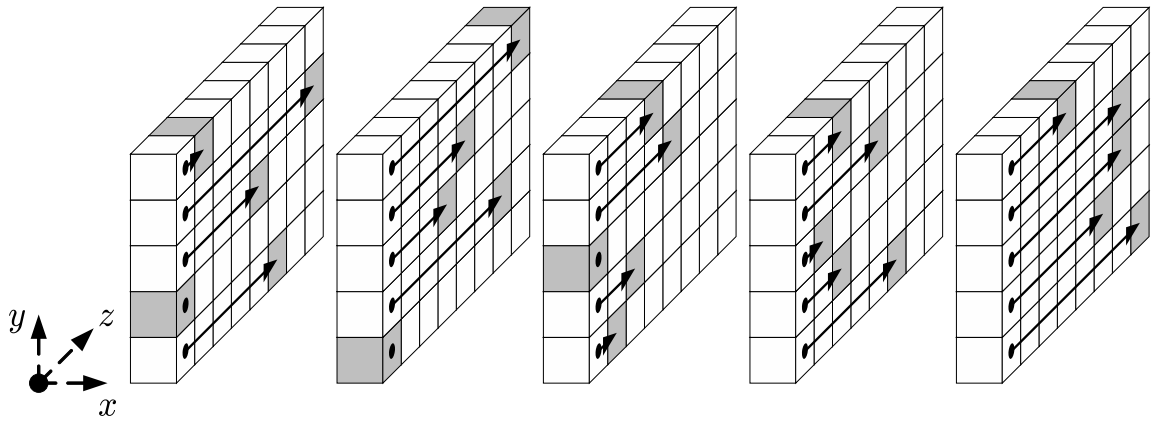
Figure 12: Applying  $\theta$  to a single bit [BDPVA11b, Figure 2.2].

Without  $\theta$ , the  $\text{KECCAK-}f$  round function would not provide diffusion of any significance.

**Step mapping  $\rho$**  (see Figure 13):

$$A[x, y, z] \leftarrow A[x, y, z - (t + 1)(t + 2)/2], \quad (4)$$

$$\text{where } t = \begin{cases} -1 & \text{if } x = y = 0, \\ \text{solution to } \begin{cases} \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}^t \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} \\ 0 \leq t \leq 23 \end{cases} & \text{in } \text{GF}(5)^{2 \times 2} \\ \end{cases} \quad \text{otherwise.} \quad (5)$$



	$x = 3$	$x = 4$	$x = 0$	$x = 1$	$x = 2$
$y = 2$	153	231	3	10	171
$y = 1$	55	276	36	300	6
$y = 0$	28	91	0	1	190
$y = 4$	120	78	210	66	253
$y = 3$	21	136	105	45	15

Figure 13: Applying  $\rho$  to the state lanes [BDPVA11b, Figure 2.4]. 5 sheets with  $w = 8$ , each of which containing 5 lanes, are shown here. The center of each slices has coordinates  $x = y = 0$ . Bottom table shows the offsets in the  $z$  direction, e.g., for  $(x, y) = (2, 0)$ , the offset is  $190 \bmod w = 6$ .

$\rho$  consists of *translations within the lanes* aimed at providing *inter-slice dispersion* [BDPVA11b, Sec. 2.3.4].

$\rho$  is *linear* and *translation-invariant* in the  $z$  direction.

Without  $\rho$ , diffusion between the slices would be slow.

**Step mapping  $\pi$**  (see Figure 14):

$$A[x, y] \leftarrow A[i, j], \quad \text{where } \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} \text{ in } \text{GF}(5)^{2 \times 2}. \quad (6)$$

🔍 Above,  $z$  index is omitted because this mapping is performed slice-wise.

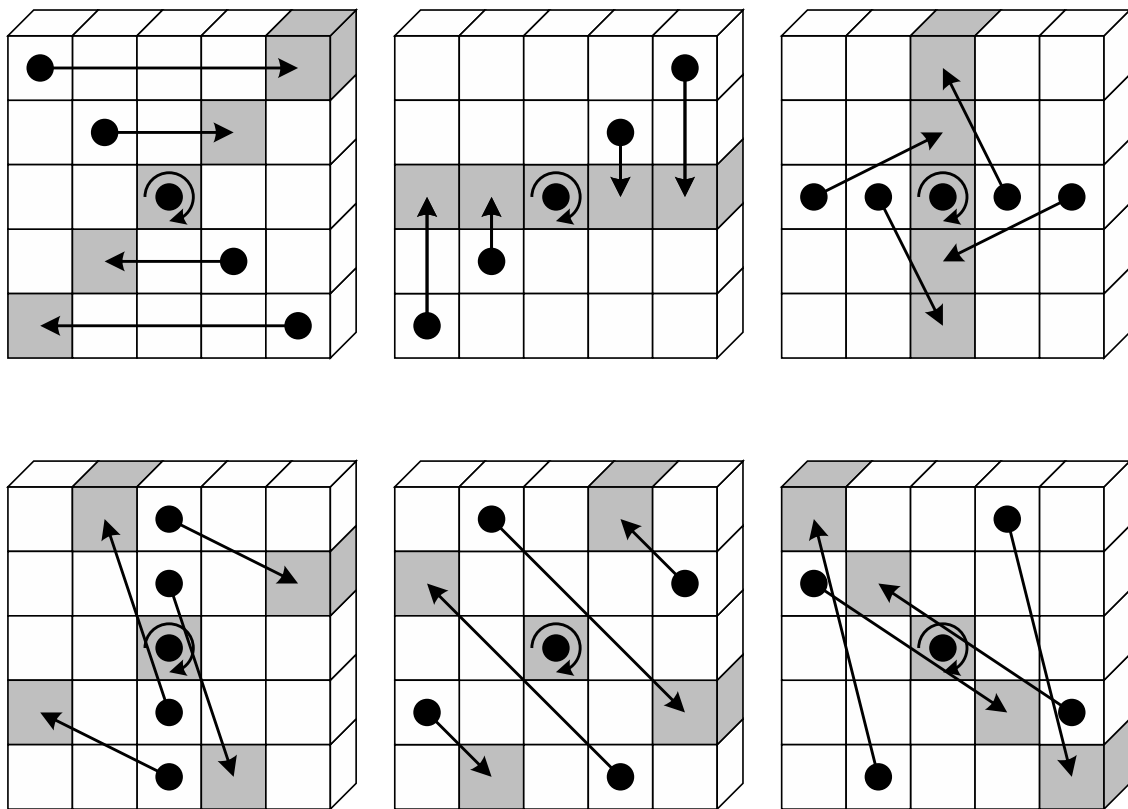


Figure 14: Applying  $\pi$  to a slice results in 24 transpositions. The lane in the origin does not change position.

$\pi$  is a linear transposition of the lanes that provides dispersion aimed at long-term diffusion.

- The lane in coordinates  $(x, y)$  is transposed to coordinates  $(x, y) \begin{bmatrix} 0 & 2 \\ 1 & 3 \end{bmatrix}$ .
- Transposition within each slice means  $\pi$  is translation-invariant in the  $z$  direction.
- Within a slice, 6 axes can be defined, where each axis defines a direction that partitions the 25 positions in a slice into 5 sets (read “4 transpositions and 1 null transposition”):
  - » Top row of [Figure 14](#)
    1.  $y = x$  axis (rising 1-slope)
    2.  $x$  axis
    3.  $y$  axis
  - » Bottom row of [Figure 14](#)
    4.  $y = -2x$  axis (falling 2-slope)
    5.  $y = 2x$  axis (rising 2-slope)
    6.  $y = -x$  axis

Without  $\pi$ ,  $\text{KECCAK-}f$  would exhibit periodic trails of low weight (which help differential and linear cryptanalysis).

**Step mapping  $\chi$**  (see [Figure 15](#)):

$$A[x] \leftarrow A[x] + (A[x + 1] + 1)A[x + 2]. \quad (7)$$

$\chi$  is the parallel applications of  $5w$  S-boxes to 5-bit rows.

$\chi$  is translation-invariant in all directions.

$\chi$  is the only nonlinear mapping in  $\text{KECCAK-}f$  [BDPVA11b, Sec. 2.3.1].

Without  $\chi$ ,  $\text{KECCAK-}f$  would be linear (which helps linear cryptanalysis).

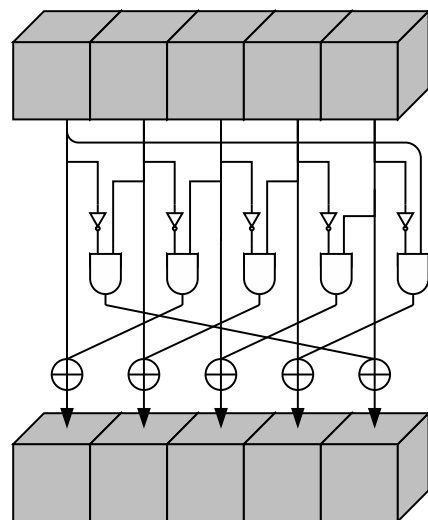


Figure 15: Applying  $\chi$  to a row [BDPVA11b, Figure 2.1].

### Step mapping $\iota$ :

$$A[x, y, z] \leftarrow A[x, y, z] + \text{RC}[i_r, x, y, z], \quad (8)$$

$$i_r = 0, \dots, n_r - 1,$$

$$n_r = 2\ell + 12, \quad (9)$$

$$\text{RC}[i_r, x, y, z] = \begin{cases} \text{rc}[j + 7i_r] & \text{if } x = 0, y = 0, z = 2^j - 1, 0 \leq j \leq \ell, \\ 0 & \text{otherwise,} \end{cases} \quad (10)$$

$$\text{rc}[t] = (x^t \bmod x^8 + x^6 + x^5 + x^4 + 1) \bmod x \text{ in GF}(2). \quad (11)$$

Above,  $i_r$  denotes the current round iteration,  $\text{RC}$  denotes the 4D matrix containing the round constants, and  $\text{rc}$  is the output of linear feedback shift register.

$\iota$  consists of the addition of round constants and is aimed at disrupting symmetry [BDPVA11b, Sec. 2.3.5].

- Round constants are added in a single lane of the state, but the disruptive effect is diffused through  $\theta$  and  $\chi$  to all lanes of the state after one round.
- The number of active bit positions of the round constants, i.e., the bit positions in which the round constant is nonzero, is  $\ell + 1$ .
- The higher  $\ell$  is, the more asymmetry is added by the round constants.

Without  $\iota$ , the round function would be translation-invariant in the  $z$  direction and all rounds would be equal making  $\text{KECCAK-}f$  subject to attacks exploiting symmetry such as slide attacks.

## 5 References

- [BR93] M. BELLARE and P. ROGAWAY, Random oracles are practical: A paradigm for designing efficient protocols, in *Proceedings of the 1st ACM Conference on Computer and Communications Security, CCS '93*, Association for Computing Machinery, New York, NY, USA, 1993, p. 62–73. <https://doi.org/10.1145/168588.168596>.

- [BDH<sup>+</sup>22] G. BERTONI, J. DAEMEN, S. HOFFERT, M. PEETERS, G. V. ASSCHE, and R. V. KEER, Glossary, TeamKeccak website, 2022, Accessed 26 Oct 2022. Available at <https://keccak.team/glossary.html>.
- [BDPVA07] G. BERTONI, J. DAEMEN, M. PEETERS, and G. VAN ASSCHE, Sponge functions, in *ECRYPT Hash Workshop*, 2007, 2007. Available at <http://sponge.noekeon.org/SpongeFunctions.pdf>.
- [BDPVA11a] G. BERTONI, J. DAEMEN, M. PEETERS, and G. VAN ASSCHE, Cryptographic sponge functions, 2011. Available at <https://keccak.team/files/CSF-0.1.pdf>.
- [BDPVA11b] G. BERTONI, J. DAEMEN, M. PEETERS, and G. VAN ASSCHE, The keccak reference, 2011, Version 3.0. Available at <https://keccak.team/files/Keccak-reference-3.0.pdf>.
- [BDPVA12] G. BERTONI, J. DAEMEN, M. PEETERS, and G. VAN ASSCHE, Duplexing the sponge: Single-pass authenticated encryption and other applications, in *Selected Areas in Cryptography* (A. MIRI and S. VAUDENAY, eds.), Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 320–337. [https://doi.org/10.1007/978-3-642-28496-0\\_19](https://doi.org/10.1007/978-3-642-28496-0_19).
- [BRS02] J. BLACK, P. ROGAWAY, and T. SHRIMPTON, Black-box analysis of the block-cipher-based hash-function constructions from pgv, in *Advances in Cryptology — CRYPTO 2002* (M. YUNG, ed.), Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 320–335.
- [Bou15] C. BOUTIN, NIST Releases SHA-3 Cryptographic Hash Standard, Press Release, 2015. Available at <https://www.nist.gov/news-events/news/2015/08/nist-releases-sha-3-cryptographic-hash-standard>.
- [Dwo15] M. J. DWORKIN, SHA-3 standard: Permutation-based hash and extendable-output functions, NIST FIPS PUB 202, August 2015. <https://doi.org/10.6028/NIST.FIPS.202>.
- [FOPS04] E. FUJISAKI, T. OKAMOTO, D. POINTCHEVAL, and J. STERN, RSA-OAEP Is Secure under the RSA Assumption, *Journal of Cryptology* 17 no. 2 (2004), 81–104. <https://doi.org/10.1007/s00145-002-0204-y>.
- [Jou09] A. JOUX, *Algorithmic Cryptanalysis*, CRC Press, 2009.
- [KL21] J. KATZ and Y. LINDELL, *Introduction to Modern Cryptography*, 3rd ed., CRC Press, 2021. Available at <https://ebookcentral.proquest.com/lib/unisa/detail.action?docID=6425020>.
- [KR11] L. R. KNUDSEN and M. J. ROBshaw, *The Block Cipher Companion, Information Security and Cryptography Texts and Monographs*, Springer Berlin Heidelberg, 2011. <https://doi.org/10.1007/978-3-642-17342-4>.
- [KM15] N. KOBLITZ and A. J. MENEZES, The random oracle model: a twenty-year retrospective, *Designs, Codes and Cryptography* 77 no. 2 (2015), 587–610. <https://doi.org/10.1007/s10623-015-0094-2>.
- [LRW02] M. LISKOV, R. L. RIVEST, and D. WAGNER, Tweakable block ciphers, in *Advances in Cryptology — CRYPTO 2002* (M. YUNG, ed.), Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 31–46.
- [LFF<sup>+</sup>15] P. LUO, Y. FEI, X. FANG, A. A. DING, D. R. KAELI, and M. LEESER, Side-channel analysis of MAC-Keccak hardware implementations, in *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy, HASP ’15*, Association for Computing Machinery, 2015. <https://doi.org/10.1145/2768566.2768567>.
- [MNS11] F. MENDEL, T. NAD, and M. SCHLÄFFER, Finding SHA-2 characteristics: Searching through a minefield of contradictions, in *Advances in Cryptology – ASIACRYPT 2011* (D. H. LEE and X. WANG, eds.), Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 288–307.
- [MvV96] A. J. MENEZES, P. C. VAN OORSCHOT, and S. A. VANSTONE, *Handbook of Applied Cryptography*, CRC Press, 1996. Available at <https://cacr.uwaterloo.ca/hac/>.
- [NIS15] NIST, Secure Hash Standard (SHS), Federal Information Processing Standards Publication 180-4, August 2015. <https://doi.org/10.6028/NIST.FIPS.180-4>.

- [SD18] K. STOFFELEN and J. DAEMEN, Column parity mixers, *IACR Transactions on Symmetric Cryptology* **2018** no. 1 (2018), 126–159. <https://doi.org/10.13154/tosc.v2018.i1.126-159>.
- [vJ11] H. C. VAN TILBORG and S. JAJODIA (eds.), *Encyclopedia of Cryptography and Security*, Springer, Boston, MA, 2011. <https://doi.org/10.1007/978-1-4419-5906-5>.