Cryptography Toolbox

So far:

stream ciphers

block ciphers

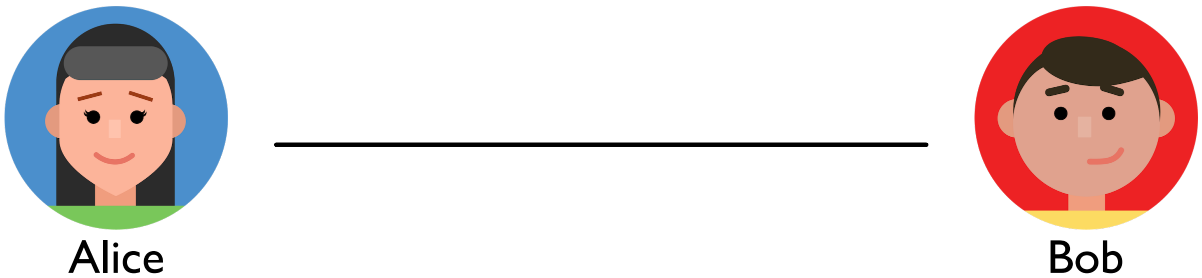
These provide **confidentiality**, but not **integrity**

Today:

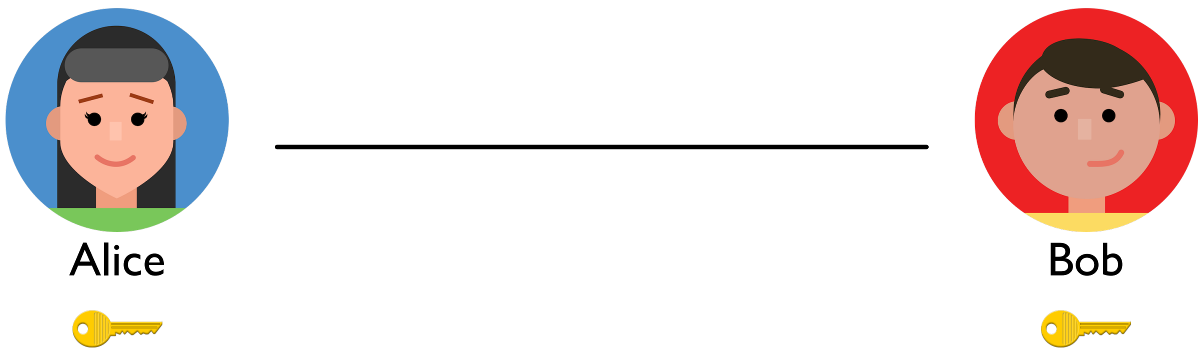
cryptographic hash functions

This is a key tool for **integrity**

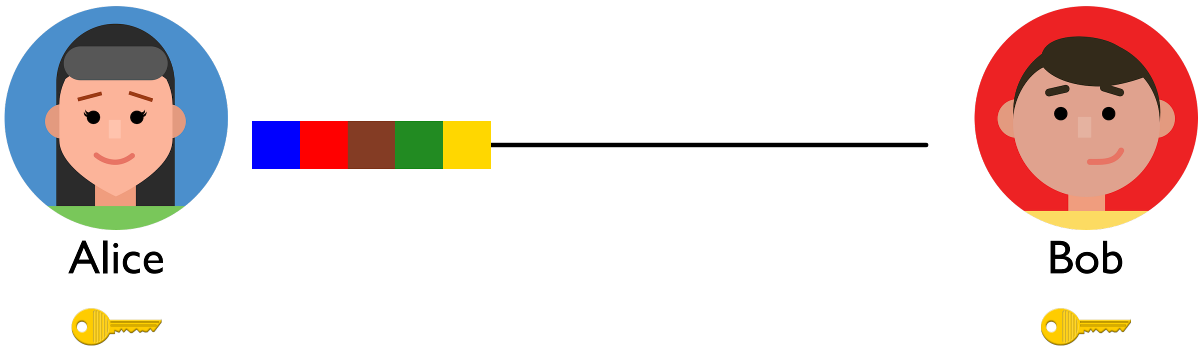
Message Integrity



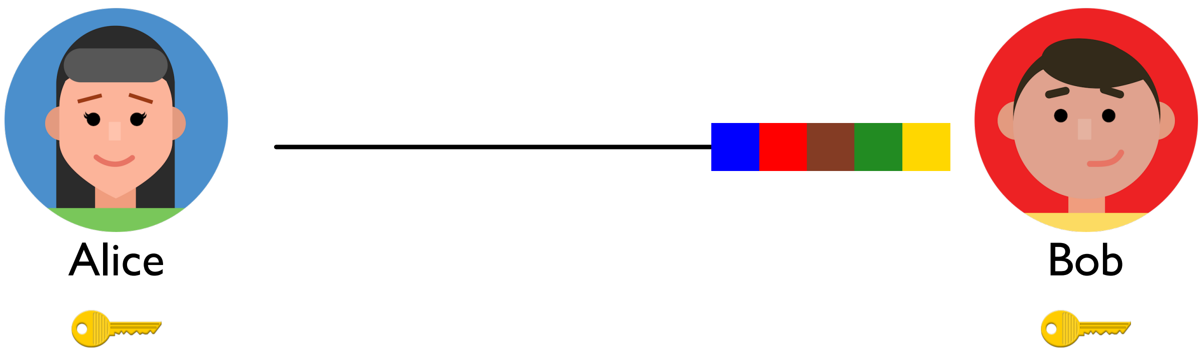
Message Integrity



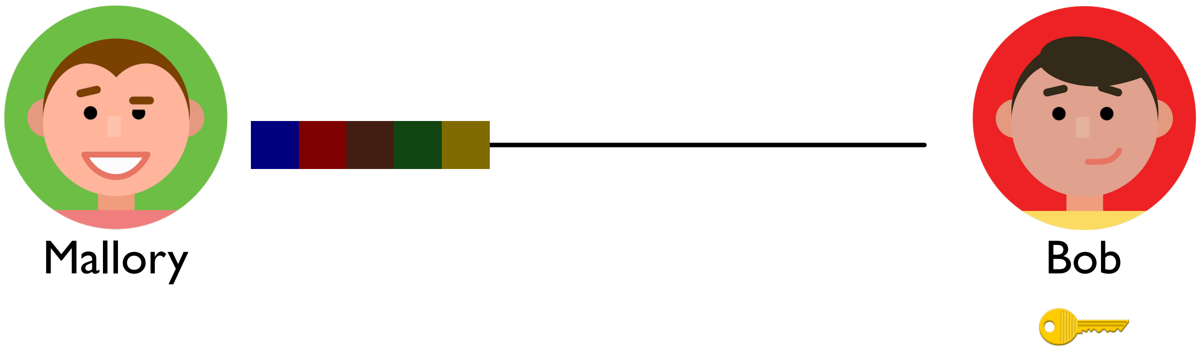
Message Integrity



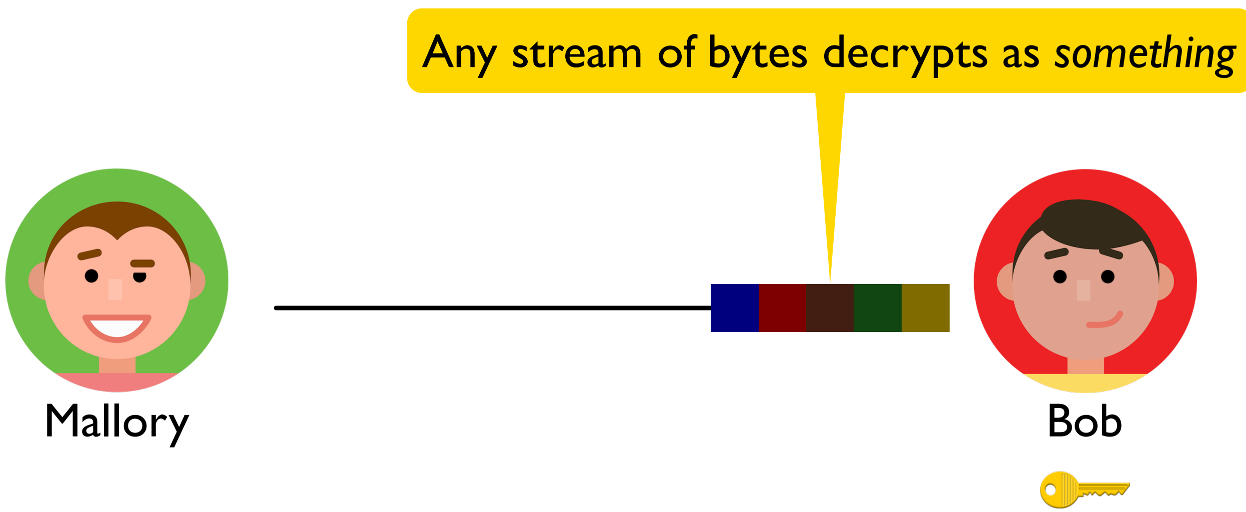
Message Integrity



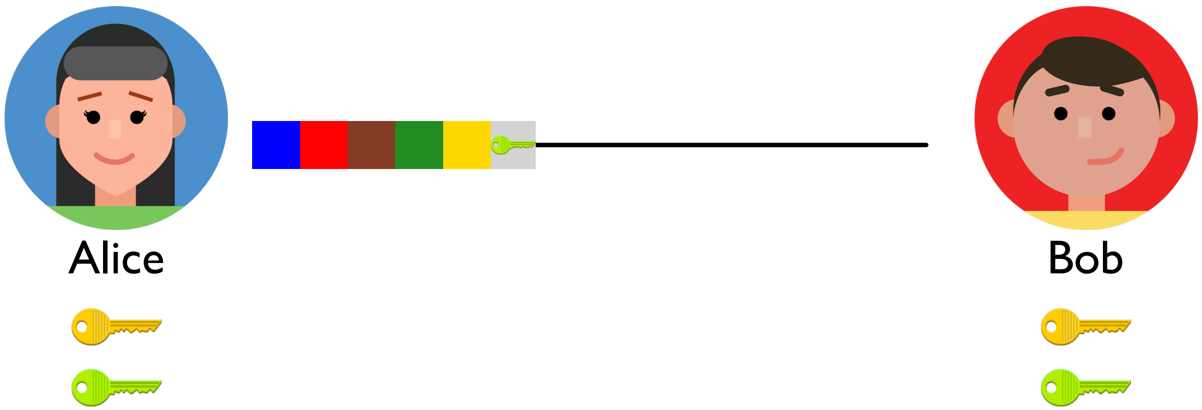
Message Integrity



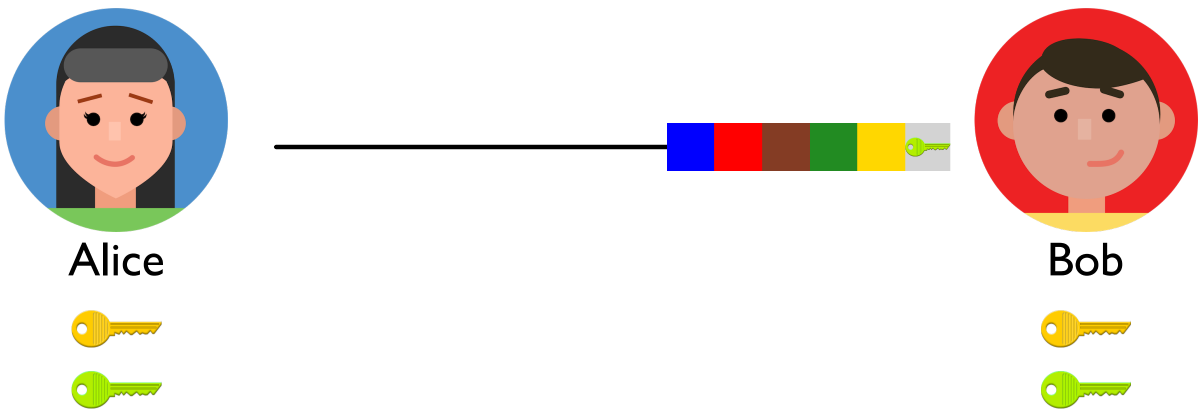
Message Integrity



Message Integrity

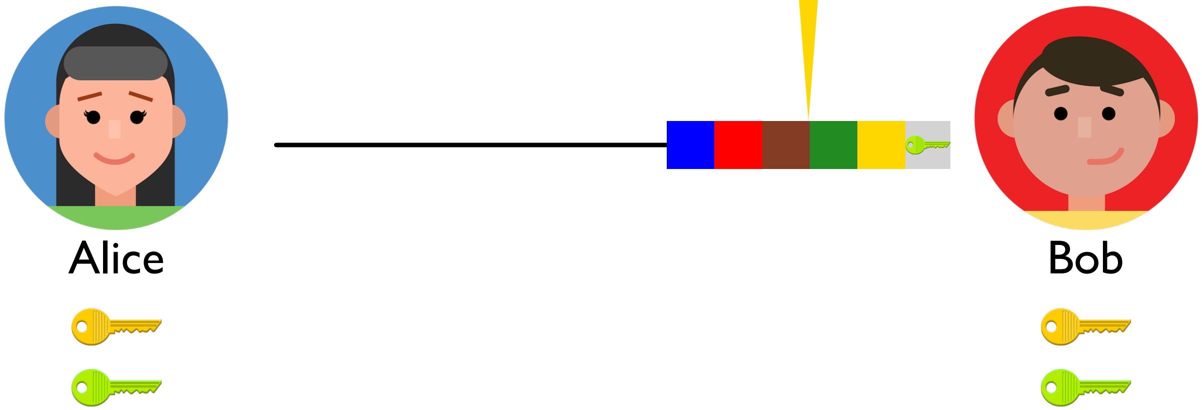


Message Integrity

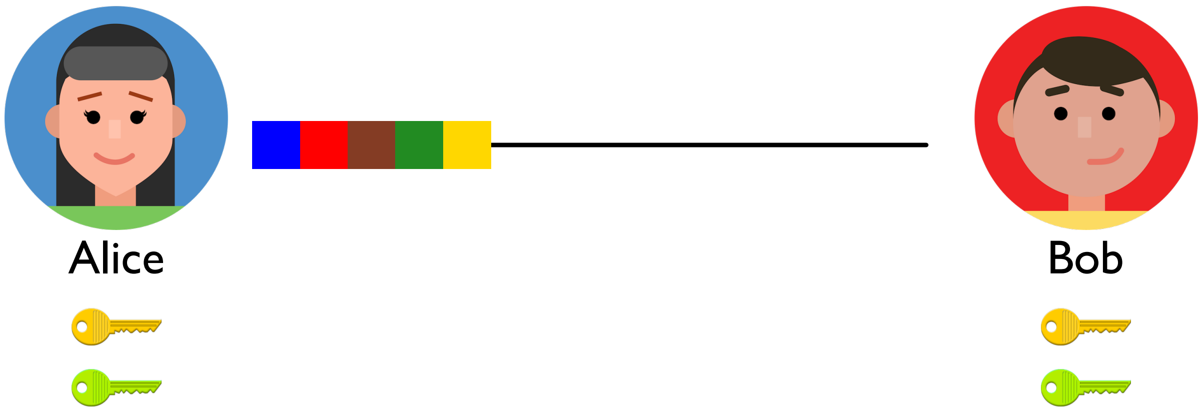


Message Integrity

Putting the key in just one place works with a **mode of operation**, but doesn't help with a stream cipher

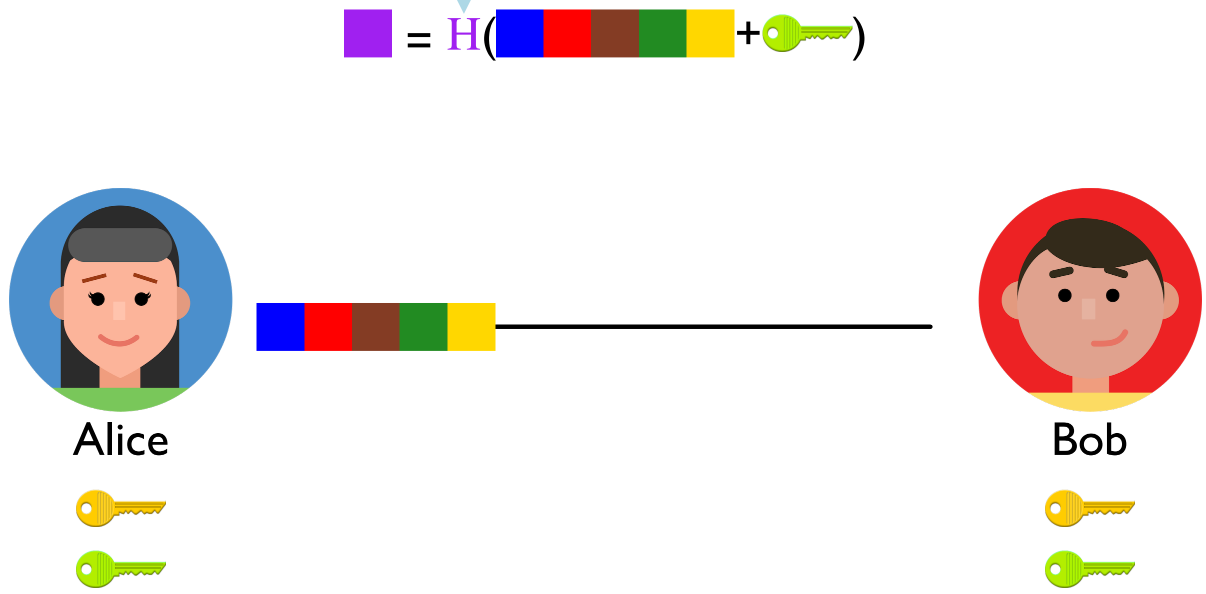


Message Integrity

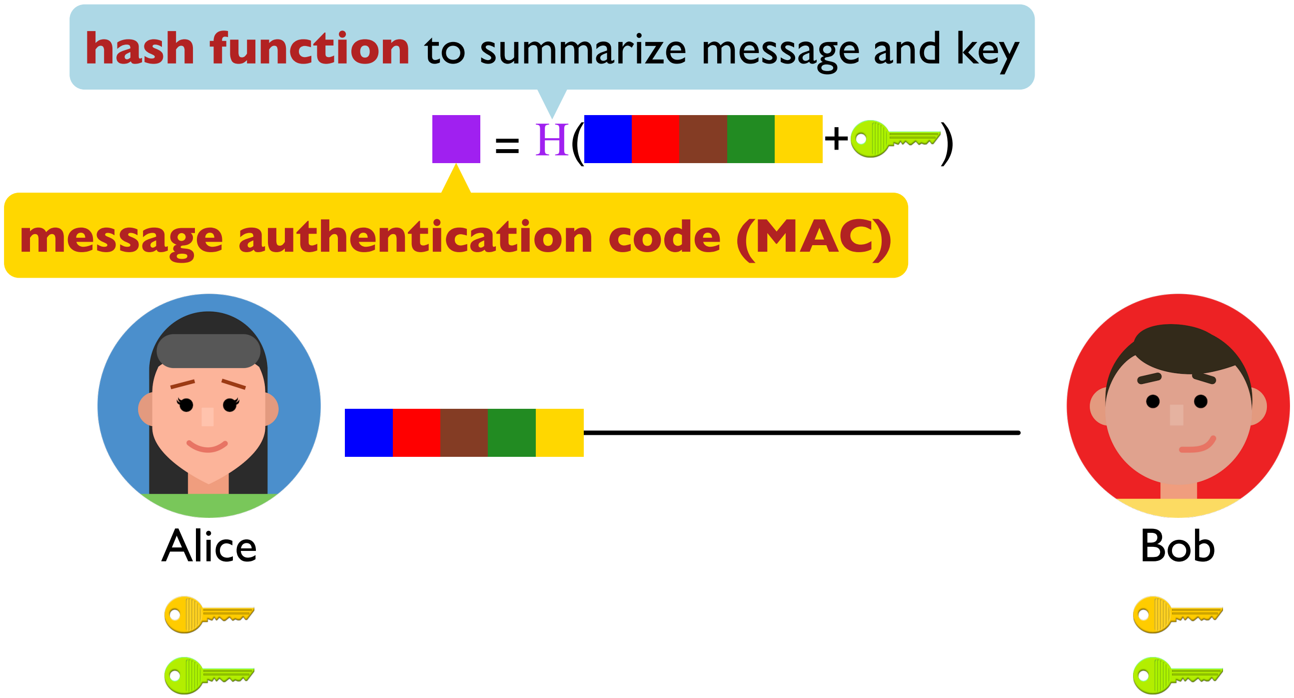


Message Integrity

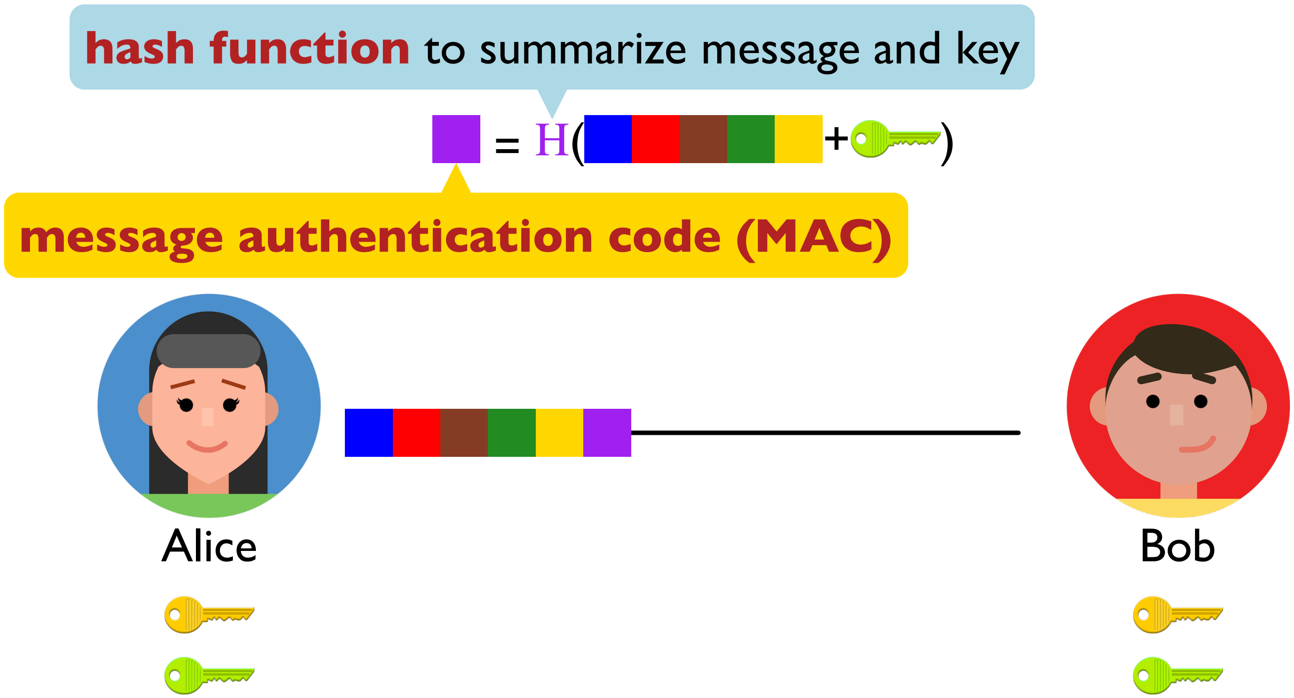
hash function to summarize message and key



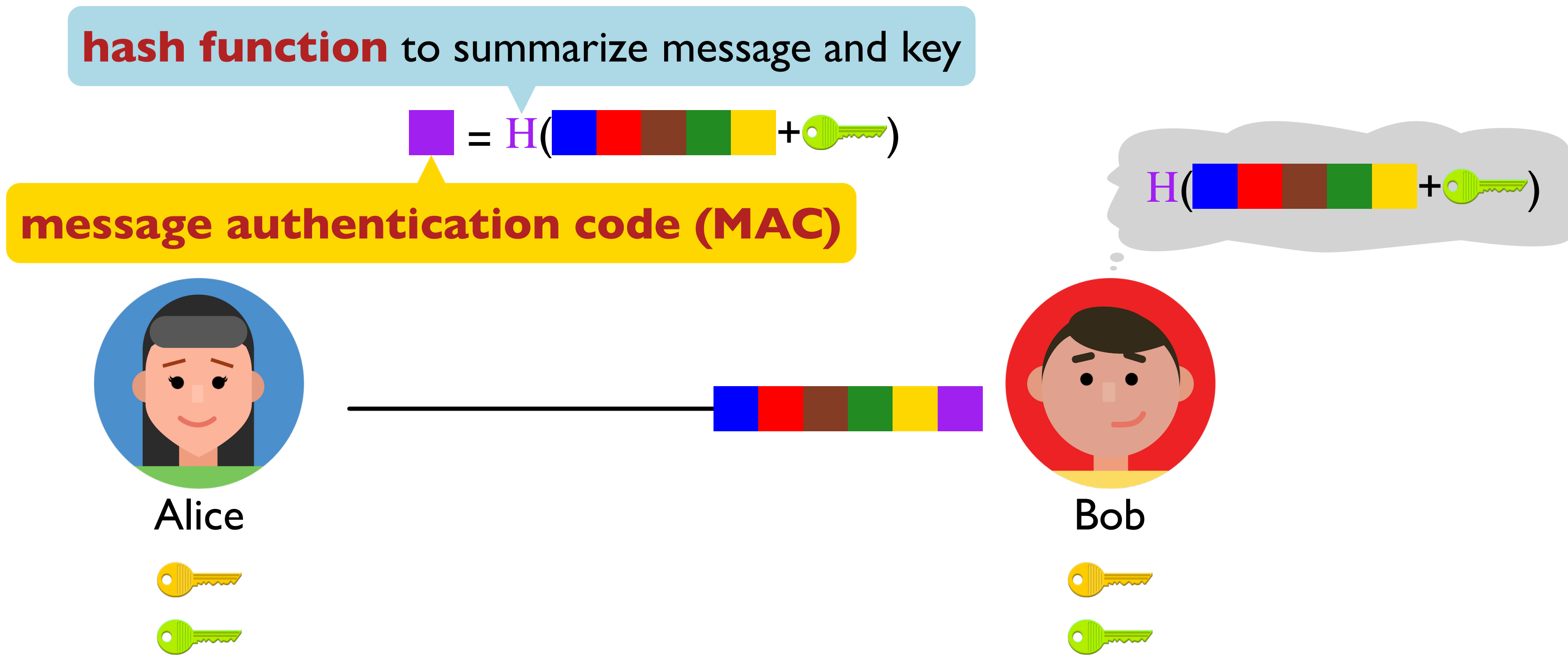
Message Integrity



Message Integrity



Message Integrity



Hash Functions

A **hash function** H maps an arbitraily large value to a fixed-sized number

Data-structure usage: fast location of a value

- Use a number an an index into an array
- Collisions are inevitable

Hash Functions

A **hash function** H maps an arbitraily large value to a fixed-sized number

Data-structure usage: fast location of a value

- Use a number an an index into an array
- Collisions are inevitable

Cryptography usage: compact representation of a value

- Use a number as a proxy, potentially hiding the original value
- Collisions should be impossible

Hash Functions

A **hash function** H maps an arbitraily large value to a fixed-sized number

Data-structure usage: fast location of a value

- Use a number an an index into an array
- Collisions are inevitable

Cryptography usage: compact representation of a value

- Use a number as a proxy, potentially hiding the original value
- Collisions should be impossible

$$H(x) = H(y) \quad \Rightarrow \quad x = y$$

Hash Functions

A **hash function** H maps an arbitraily large value to a fixed-sized number

Data-structure usage: fast location of a value

- Use a number an an index into an array
- Collisions are inevitable

Cryptography usage: compact representation of a value

- Use a number as a proxy, potentially hiding the original value
- Collisions should be impossible

How is this possible?

Hash Functions

A **hash function** H maps an arbitraily large value to a fixed-sized number

Data-structure usage: fast location of a value

- Use a number an an index into an array
- Collisions are inevitable

Cryptography usage: compact representation of a value

- Use a number as a proxy, potentially hiding the original value
- Collisions should be *infeasible*

Hash Collisions

If you have a hash array of length 256 and a ideal hash function H, how many items until you expect to find a collision?

Probably that 2 items *don't* collide: $\frac{255}{256} = 99.6\%$

Probably that 3 items *don't* collide: $\frac{255}{256} \times \frac{254}{256} = 98.8\%$

Probably that 4 items *don't* collide: $\frac{255}{256} \times \frac{254}{256} \times \frac{253}{256} = 97.6\%$

Hash Collisions

If you have a hash array of length 256 and a ideal hash function H, how many items until you expect to find a collision?

Probability of no collisions:

1	100.0%	11	80.4%	21	43.0%
2	99.6%	12	76.9%	22	39.5%
3	98.8%	13	73.3%	23	36.1%
4	97.6%	14	69.6%	24	32.8%
5	96.1%	15	65.8%	25	29.7%
6	94.2%	16	61.9%	26	26.8%
7	92.0%	17	58.0%	27	24.1%
8	89.5%	18	54.2%	28	21.6%
9	86.7%	19	50.4%	29	19.2%
10	83.6%	20	46.6%	30	17.0%

Hash Collisions

If you have a hash array of length 256 and a ideal hash function H, how many items until you expect to find a collision?

Probability of no collisions:

1	100.0%	11	80.4%	21	43.0%
2	99.6%	12	76.9%	22	39.5%
3	98.8%	13	73.3%	23	36.1%
4	97.6%	14	69.6%	24	32.8%
5	96.1%	15	65.8%	25	29.7%
6	94.2%	16	61.9%	26	26.8%
7	92.0%	17	58.0%	27	24.1%
8	89.5%	18	54.2%	28	21.6%
9	86.7%	19	50.4%	29	19.2%
10	83.6%	20	46.6%	30	17.0%

Birthday paradox: In a room with only 23 people, probably two people in the room have the same birthday

Hash Collisions

If you have a hash array of length 2^N and a ideal hash function H, how many items until you expect to find a collision?

Probability of no collisions with k values:

$$\frac{2^N!}{2^{kN}(2^N - k)!}$$

Approximate k where probability reaches 50%:

$$2^{N/2}$$

$256 \Rightarrow N = 8 \Rightarrow k = 16$, which is in the right neighborhood

Cryptographic Hash Collisions

For cryptographic purposes, we're not allocating an array, so we can use a much larger N

Hash code bits N	Expected collsision at
128	$2^{64} = 1.8 \times 10^{19}$
256	$2^{128} = 3.4 \times 10^{38}$
512	$2^{256} = 1.2 \times 10^{77}$

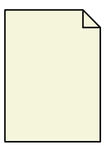
Number of atoms in the universe ≈ 10^{80}

Cryptographic Hash Assumptions

Needed for a MAC:

$$H(x) = H(y) \quad \Rightarrow \quad x = y$$

Also useful as a secure document checksum



`download_me` $\Rightarrow$ `c2c594e8d3f81db4b6a9340d5cb1903b2c9e622179ae4955d353bd54c5e3af9c`

Cryptographic Hash Assumptions

Needed for a MAC:

$$H(x) = H(y) \quad \Rightarrow \quad x = y$$

For some other purposes, we also need

given $H(x)$, cannot compute x

For example, password checks without storing passwords

Attack Modes

Known x, try to find colliding y

Example: malicious substitute for a download

Find both x and y that collide

Example: convince to accept x, later substitute y

Known $H(x)$, find x

Example: extract password from saved hash

Standardized Cryptographic Hash Functions

name	*hash bits*	*status*	*algorithm family*
MD5	128	*collisions found*	Merkle–Damgård
SHA-1	160	*some collisions found*	Merkle–Damgård
SHA-2	256 or 512	considered secure	Merkle–Damgård
SHA-3	256 or 512	considered secure	Keccak

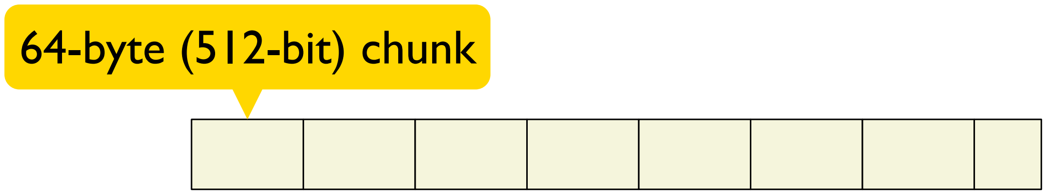
SHA-256 and SHA-512 are the 256-bit and 512-bit variants of SHA-2
SHA3-256 and SHA3-512 are the variants of SHA-3

SHA-3 is intended as a potential drop-in replacement for SHA-2 — in case a weakness in SHA-2 is discovered

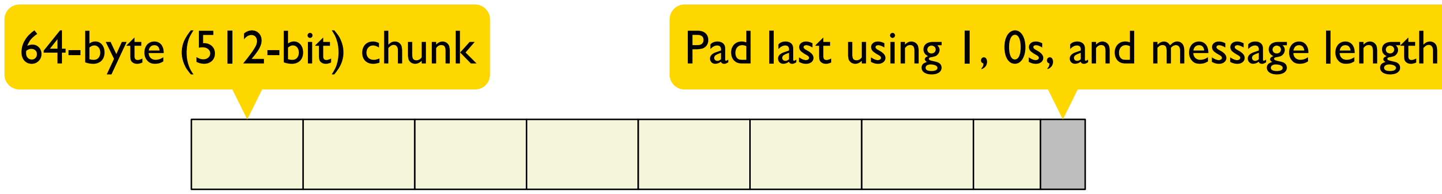
MD5

plaintext

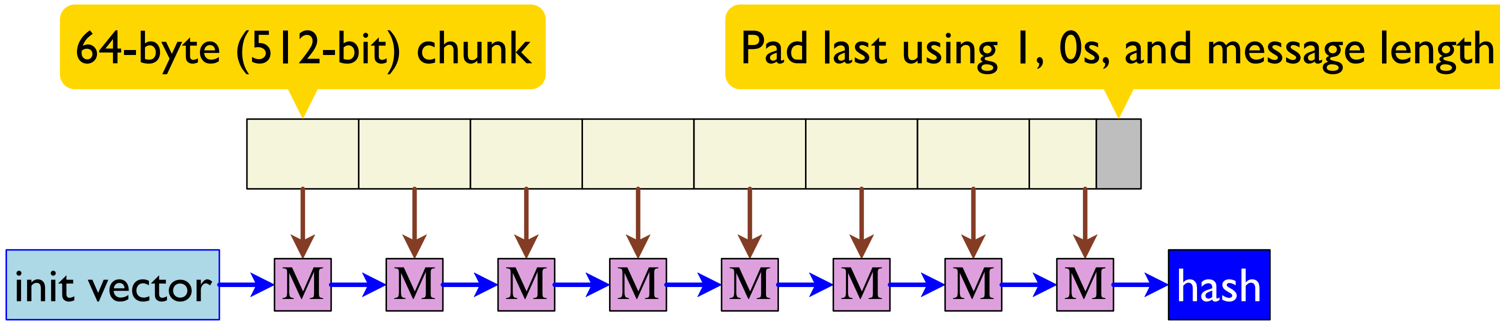
MD5



MD5

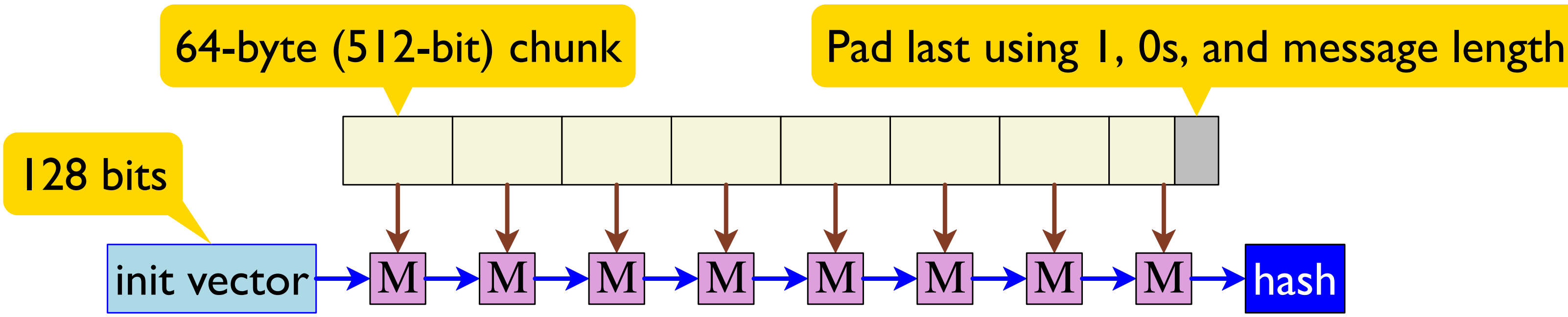


MD5

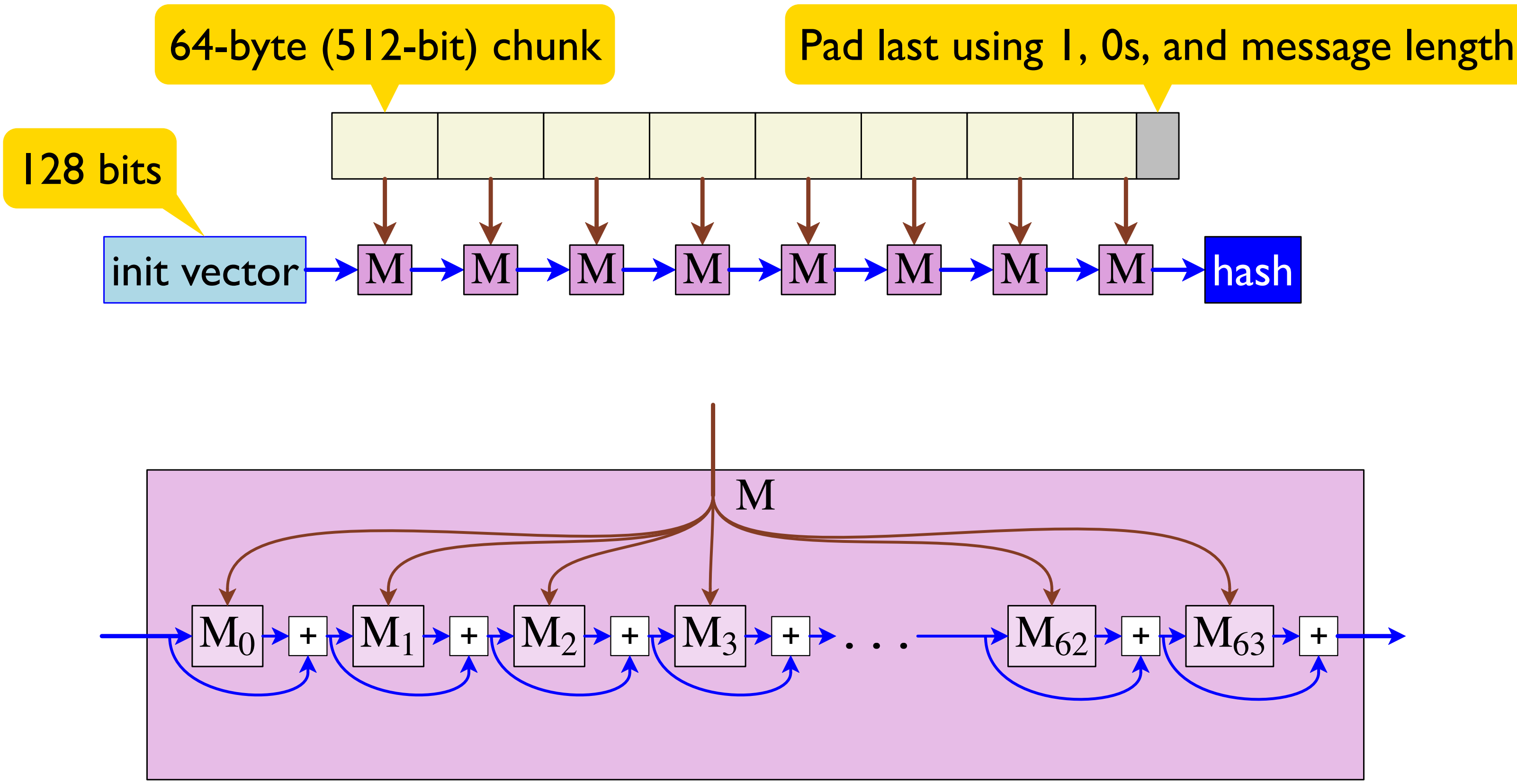


64-byte (512-bit) chunk
Pad last using 1, 0s, and message length
init vector
M
M
M
M
M
M
M
M
hash

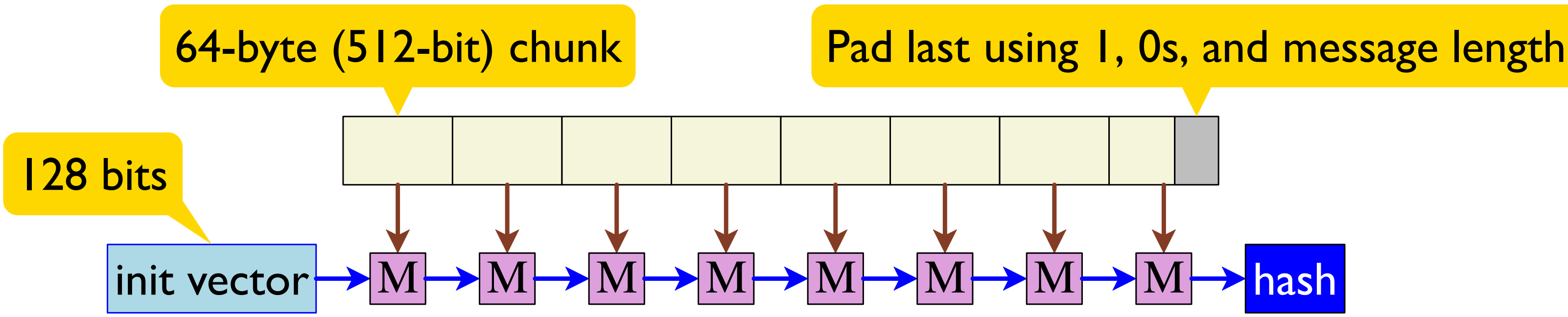
MD5


64-byte (512-bit) chunk
Pad last using 1, 0s, and message length
128 bits
init vector
M
M
M
M
M
M
M
M
hash

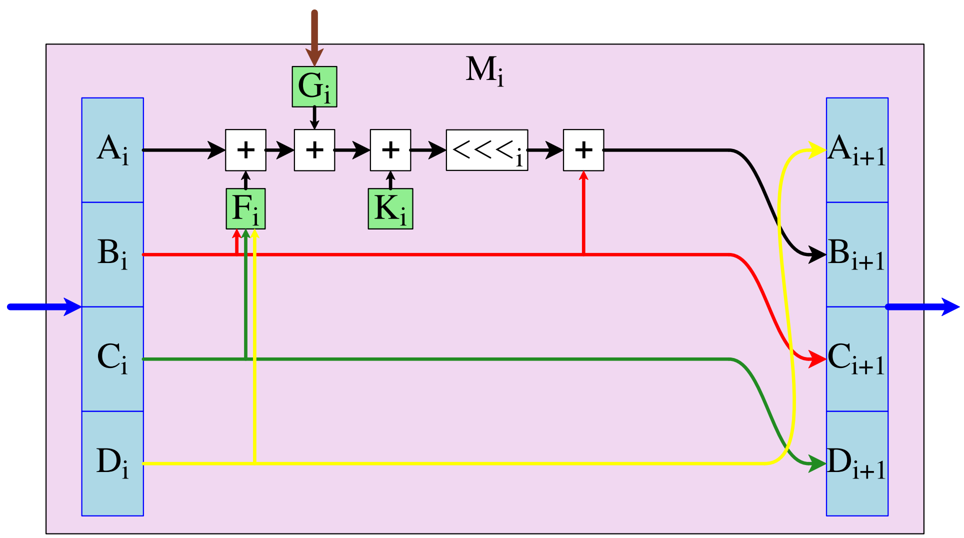
MD5


64-byte (512-bit) chunk
Pad last using 1, 0s, and message length
128 bits
init vector
M
hash
M
M_0
M_1
M_2
M_3
M_{62}
M_{63}
+
. . .

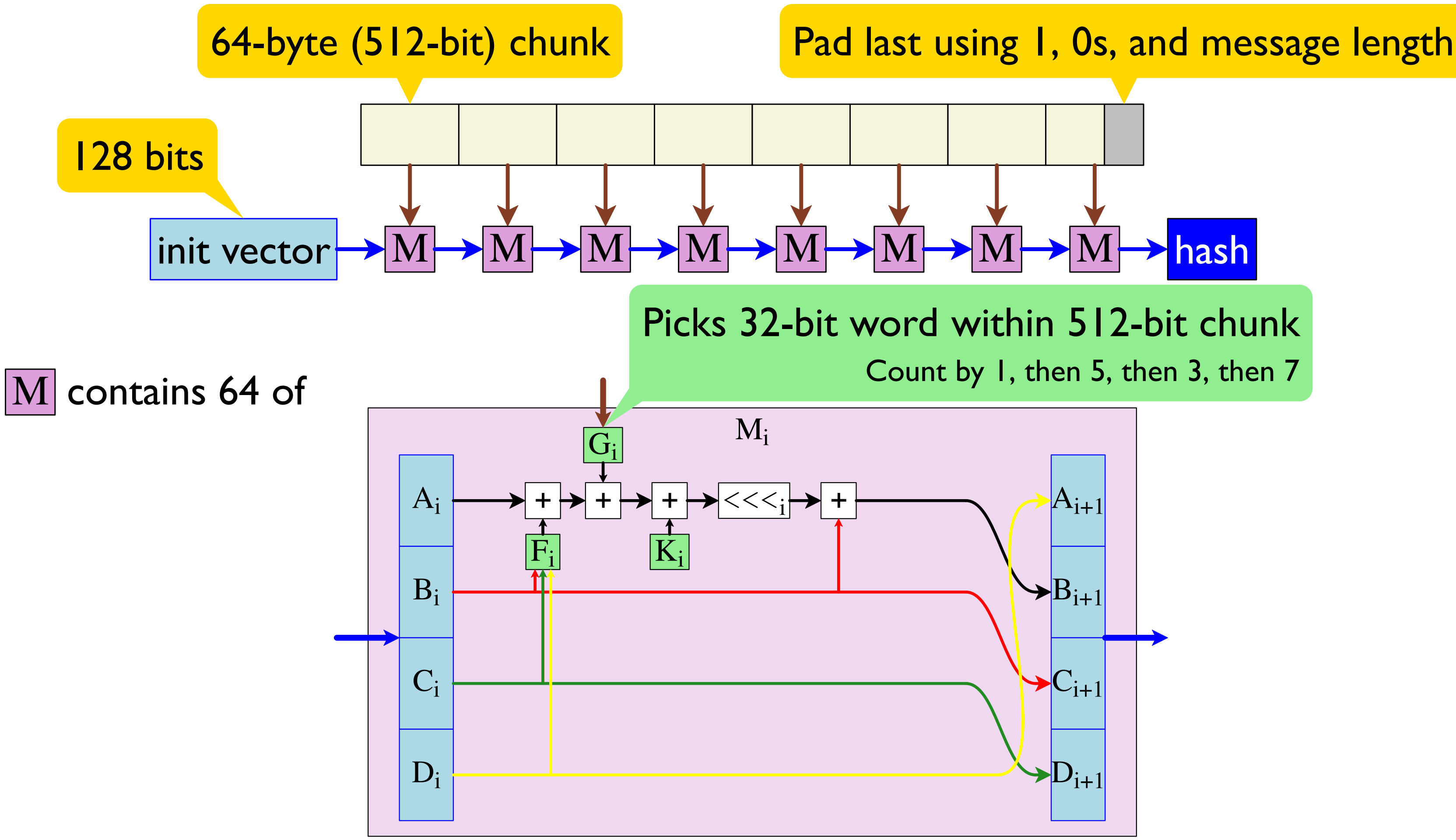
MD5



M contains 64 of

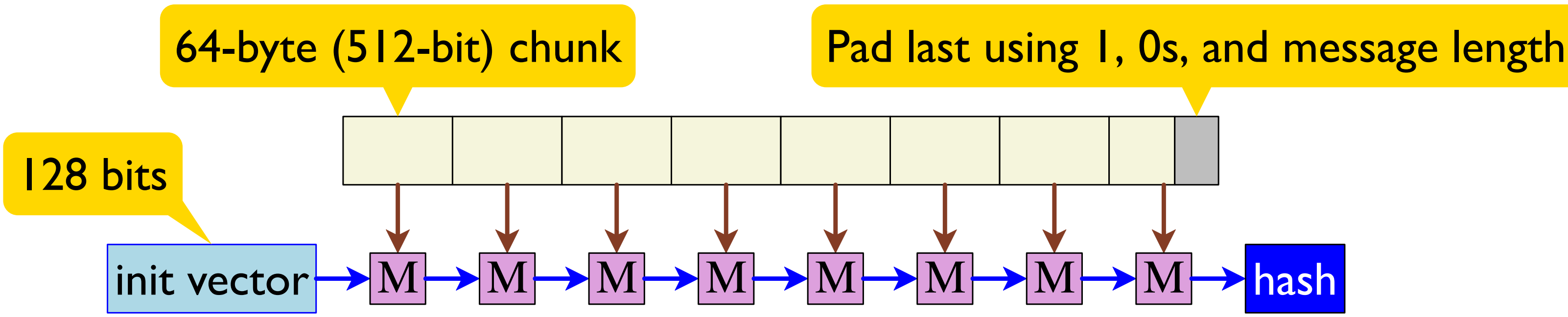


MD5

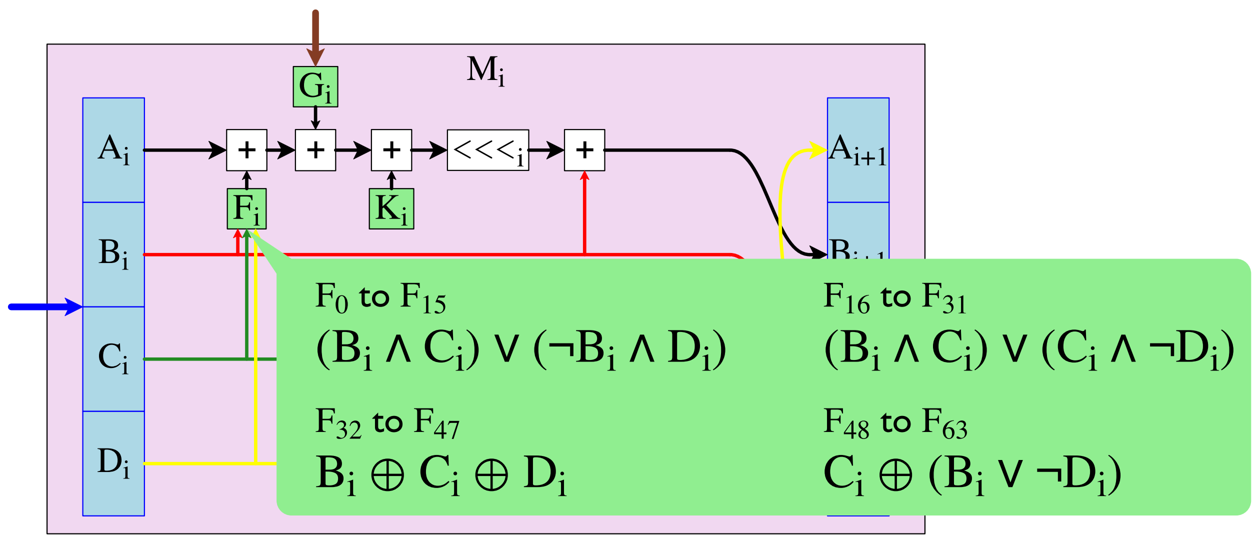


64-byte (512-bit) chunk
Pad last using 1, 0s, and message length
128 bits
init vector
M
M
M
M
M
M
M
M
hash
Picks 32-bit word within 512-bit chunk
Count by 1, then 5, then 3, then 7
M contains 64 of
Mi
Gi
Ai
Bi
Ci
Di
Fi
Ki
+
+
+
<<<i
+
Ai+1
Bi+1
Ci+1
Di+1

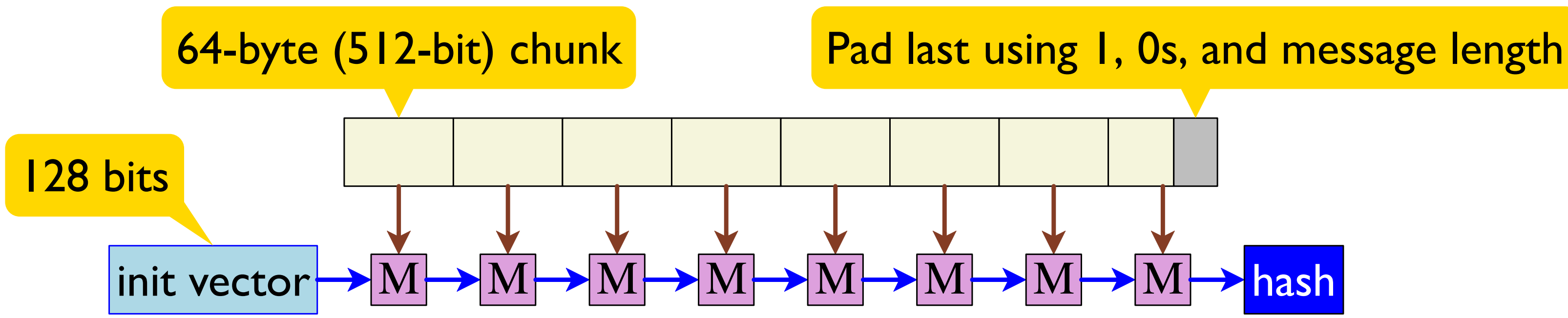
MD5



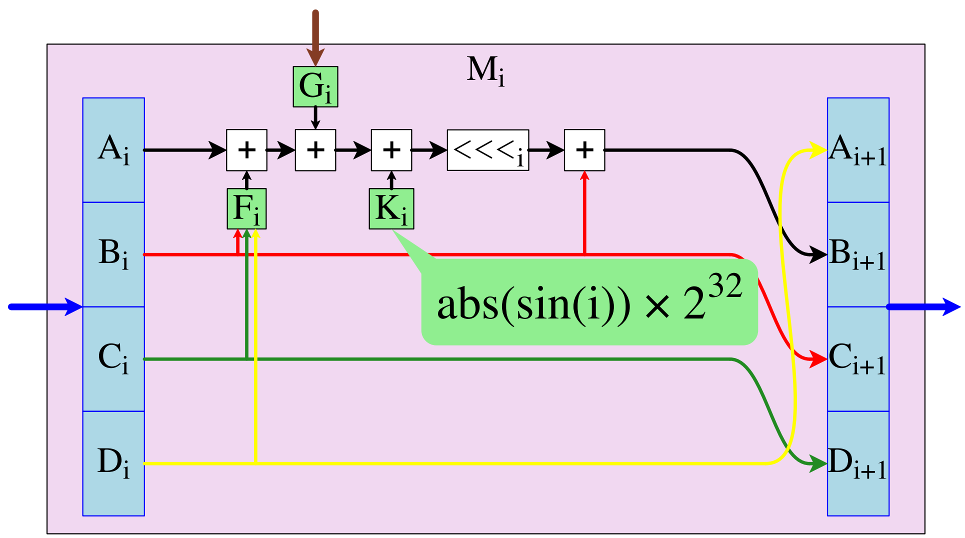
M contains 64 of



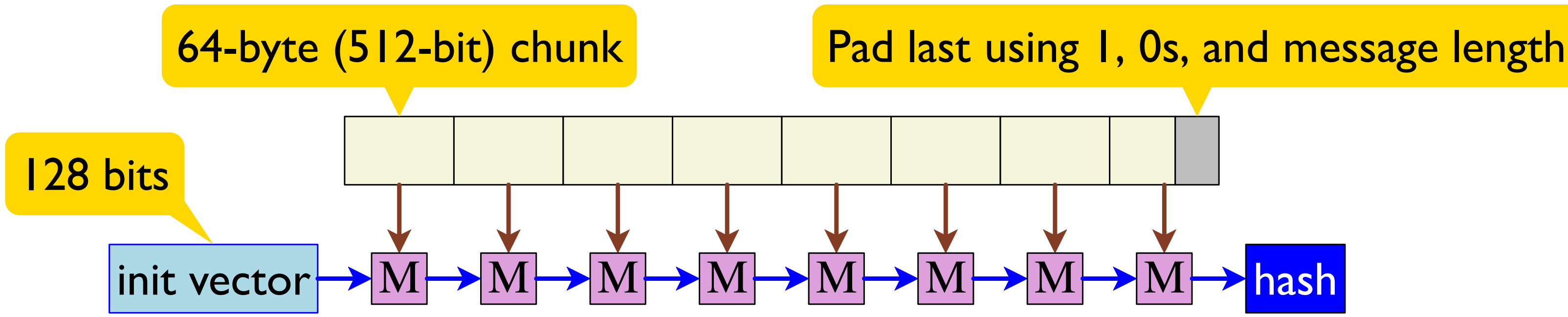
MD5

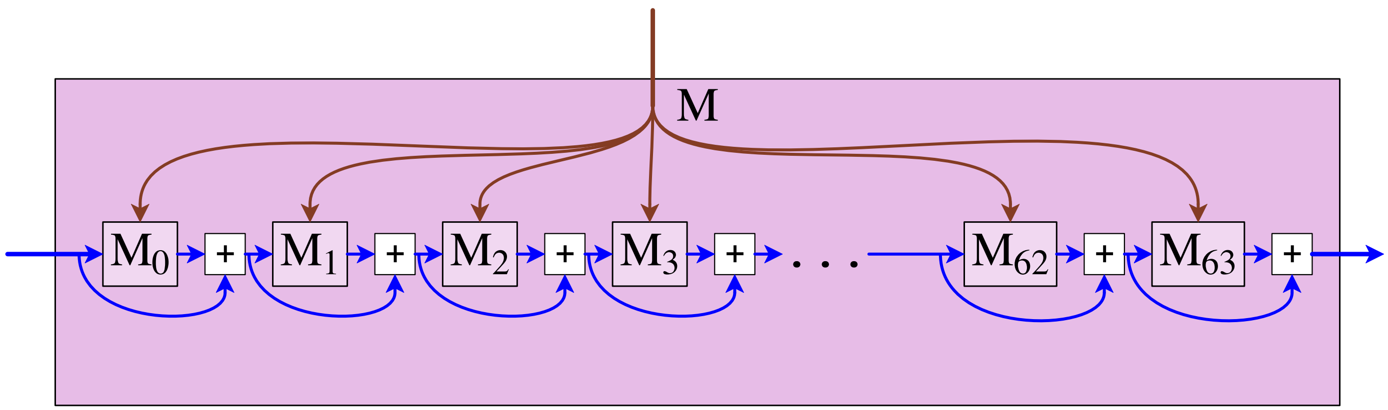


M contains 64 of



MD5


64-byte (512-bit) chunk
Pad last using 1, 0s, and message length
128 bits
init vector
M
hash


M
M_0
M_1
M_2
M_3
M_{62}
M_{63}
+

Hashing and Passwords

Servers don't want to know your password...

They want to know that you know it

Store a hash of a password, not the password:

user	H(password)
alice	d8ef3b7d2e6a8
bob	a6fdb8307dbc0
eve	9759a5d1558e4
carol	a6fdb8307dbc0

Server has to know password as you're logging in,
bit it only has to *store* a hash

Hashing and Passwords

Servers don't want to know your password...

They want to know that you know it

Store a hash of a password, not the password:

user	H(password)
alice	d8ef3b7d2e6a8
bob	a6fdb8307dbc0
eve	9759a5d1558e4
carol	a6fdb8307dbc0

Cannot reconstruct alice's password from hash

Server has to know password as you're logging in,
bit it only has to *store* a hash

Hashing and Passwords

Servers don't want to know your password...

They want to know that you know it

Store a hash of a password, not the password:

user	H(password)
alice	d8ef3b7d2e6a8
bob	a6fdb8307dbc0
eve	9759a5d1558e4
carol	a6fdb8307dbc0

Uh oh —
Can tell that bob and carol have the same password

Server has to know password as you're logging in,
bit it only has to *store* a hash

Hashing and Passwords

Don't store passwords

Don't store hashed passwords

Store a **salted hash** of a password:

user	salt	H(password+salt)
alice	adg3fee684	f3b4dd8e2e6a8
bob	992a6df99a	8307a6fbbdac0
eve	1aac7deef0	1558e49229a5d
carol	8a8721fbb1	07dbc0a99db83

Hashing and Passwords

Don't store passwords

Don't store hashed passwords

Store a **salted hash** of a password:

user	salt	H(password+salt)
alice	adg3fee684	f3b4dd8e2e6a8
bob	992a6df99a	8307a6fbbdac0
eve	1aac7deef0	1558e49229a5d
carol	8a8721fbb1	07dbc0a99db83

Randomly generated when password is set

Hashing and Passwords

Don't store passwords

Don't store hashed passwords

Store a **salted hash** of a password:

user	salt	H(password+salt)
`alice`	`adg3fee684`	`f3b4dd8e2e6a8`
`bob`	`992a6df99a`	`8307a6fbbdac0`
`eve`	`1aac7deef0`	`1558e49229a5d`
`carol`	`8a8721fbb1`	`07dbc0a99db83`

Even if `bob` **and** `carol` **both have the password** `passwd`**,**
H(`passwd`+`992a6df99a`) ≠ H(`passwd`+`8a8721fbb1`)

Summary

A **cryptographic hash function** is a one-way hash function that avoids collisions

- Useful for ensuring message integrity
- Useful for perserving evidence but forgetting details

You should use **SHA-2**

Don’t manage passwords yourself, but if you do, store only **salted hashes of passwords**