Improving Speed and Security in Updatable Encryption Schemes

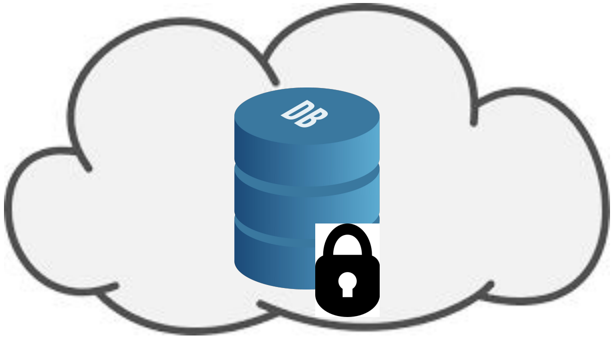
Dan Boneh
Stanford University

Saba Eskandarian
Stanford University

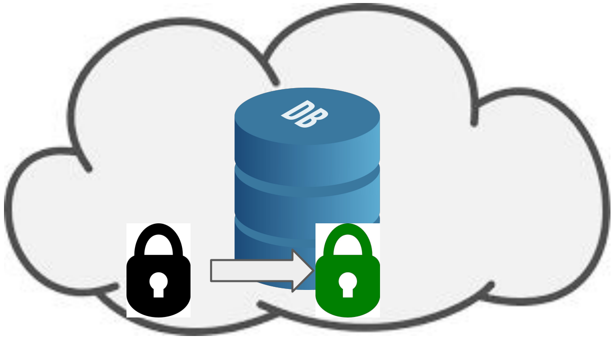
Sam Kim
Stanford University

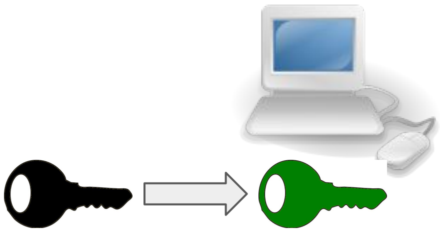
Maurice Shih
Cisco Systems

Key Rotation



Key Rotation





Good Reasons to Rotate Keys

1. Recommended by NIST (Special Publication 800-57)

Good Reasons to Rotate Keys

1. Recommended by NIST (Special Publication 800-57)

2. Recommended by Google (cloud.google.com/kms/docs/key-rotation)

Good Reasons to Rotate Keys

1. Recommended by NIST (Special Publication 800-57)
2. Recommended by Google (cloud.google.com/kms/docs/key-rotation)
3. Required by PCI DSS (PCI DSS 3.6.4)

Good Reasons to Rotate Keys

1. Recommended by NIST (Special Publication 800-57)
2. Recommended by Google (cloud.google.com/kms/docs/key-rotation)
3. Required by PCI DSS (PCI DSS 3.6.4)

…But *Why?*

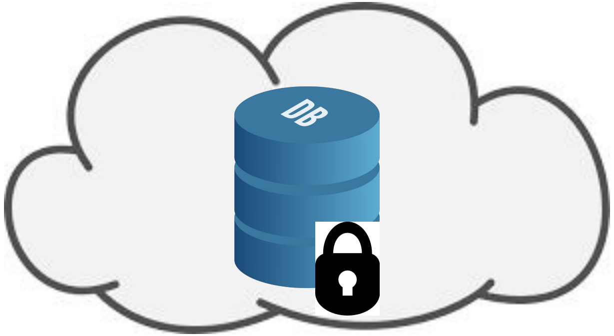
Good Reasons to Rotate Keys

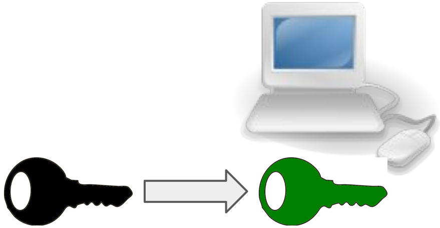
Reasons to rotate keys for data stored in the cloud:

- Compromised keys need to be taken out of use
- Proactive refresh of keys
- Access control enforcement

How to Rotate Keys in the Cloud?

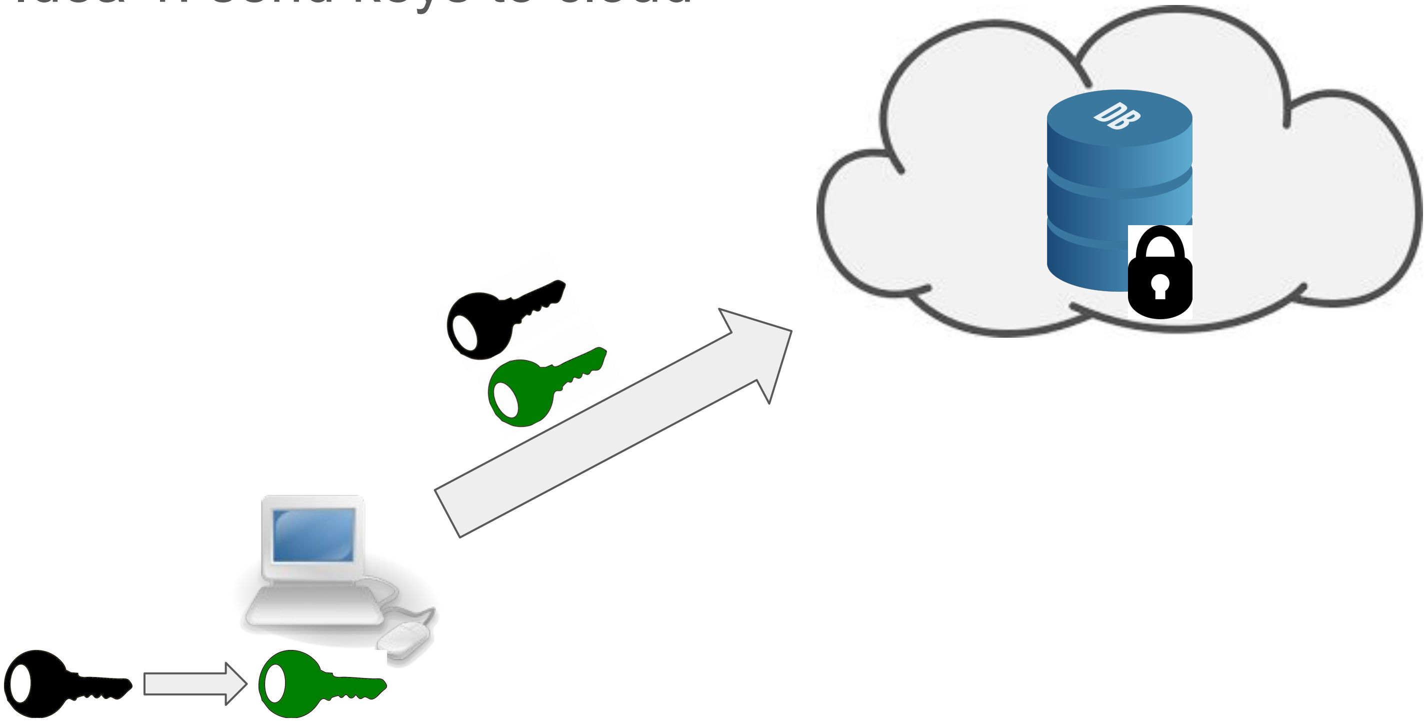
Idea 1: send keys to cloud





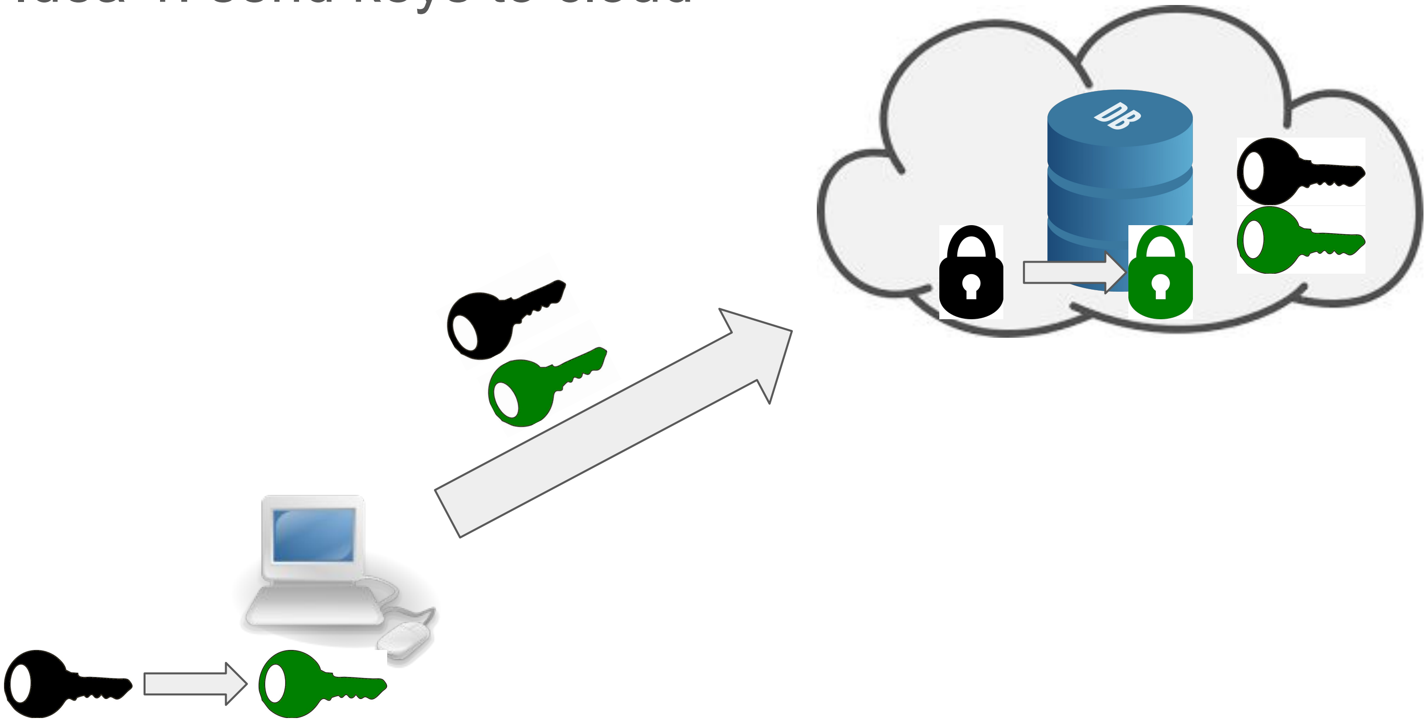
How to Rotate Keys in the Cloud?

Idea 1: send keys to cloud



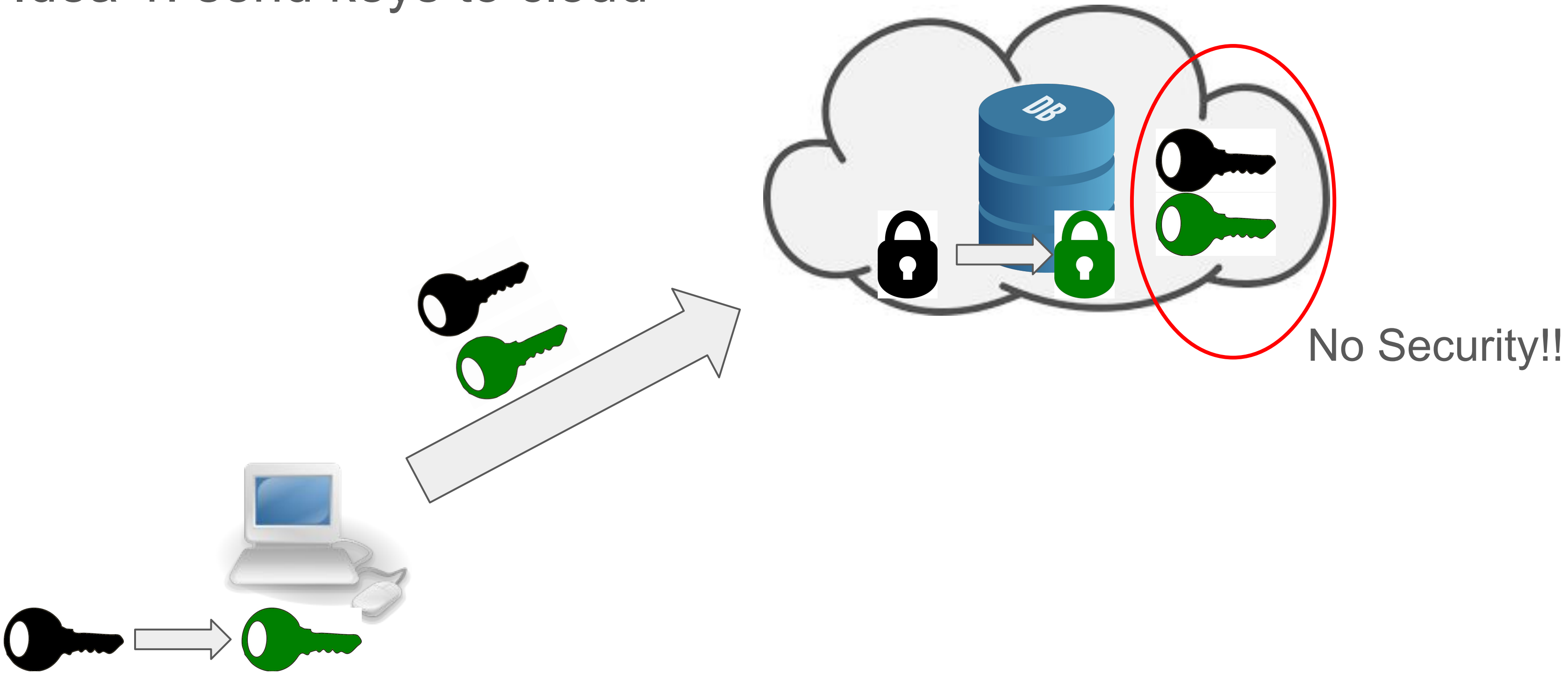
How to Rotate Keys in the Cloud?

Idea 1: send keys to cloud



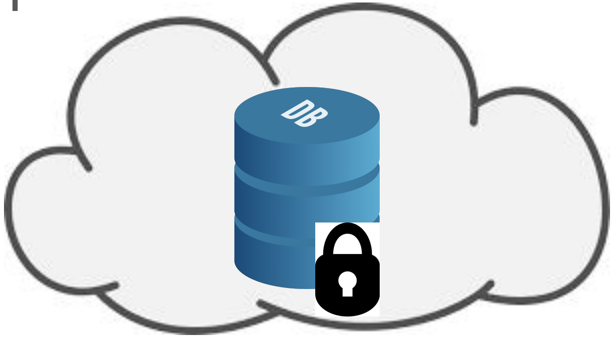
How to Rotate Keys in the Cloud?

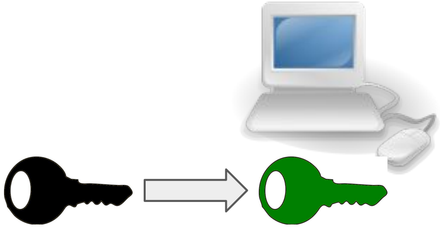
Idea 1: send keys to cloud



How to Rotate Keys in the Cloud?

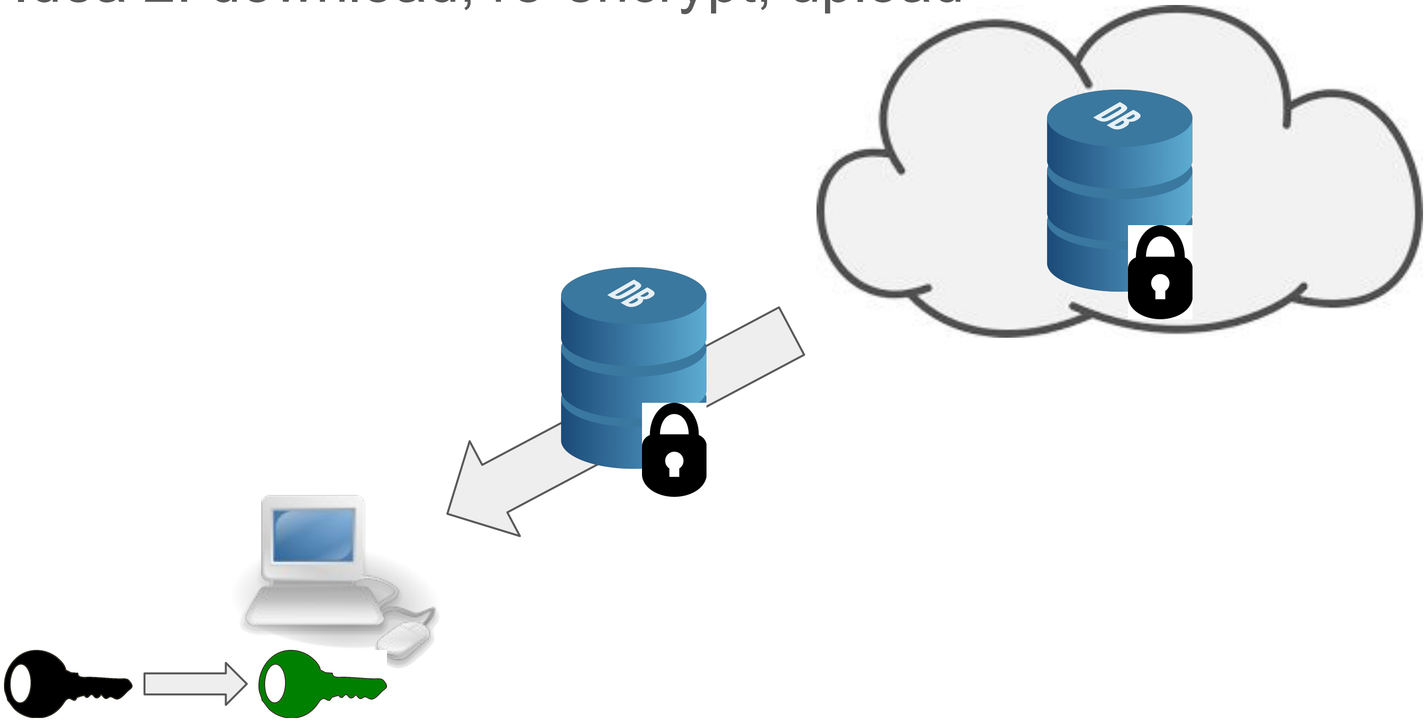
Idea 2: download, re-encrypt, upload





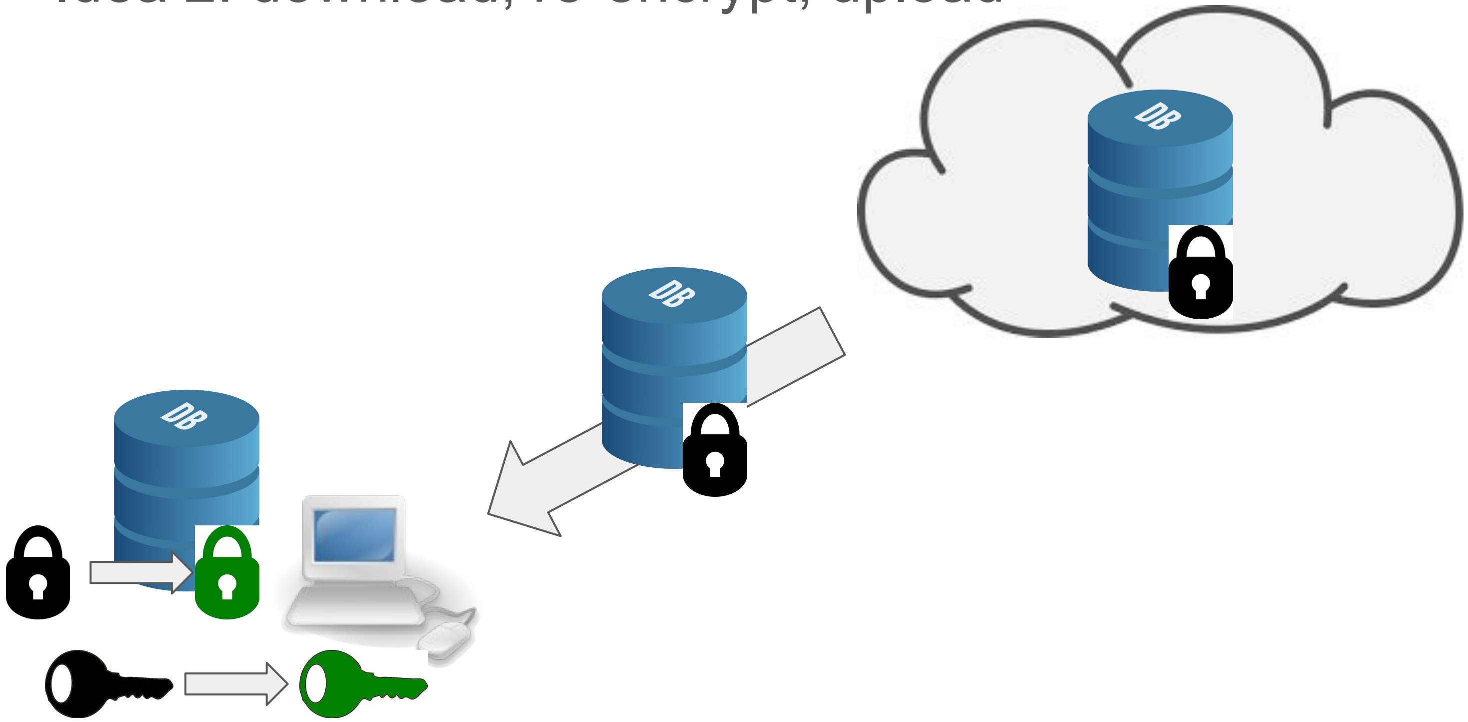
How to Rotate Keys in the Cloud?

Idea 2: download, re-encrypt, upload



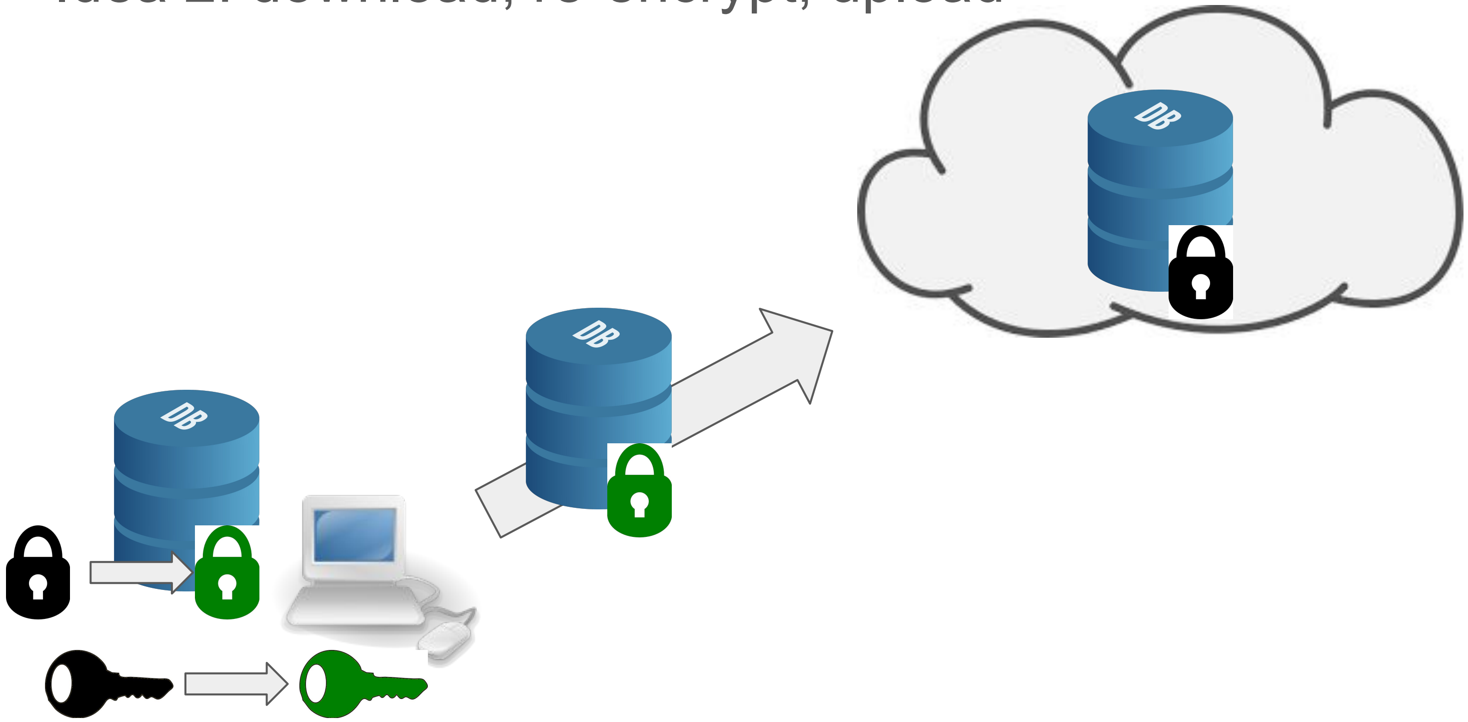
How to Rotate Keys in the Cloud?

Idea 2: download, re-encrypt, upload



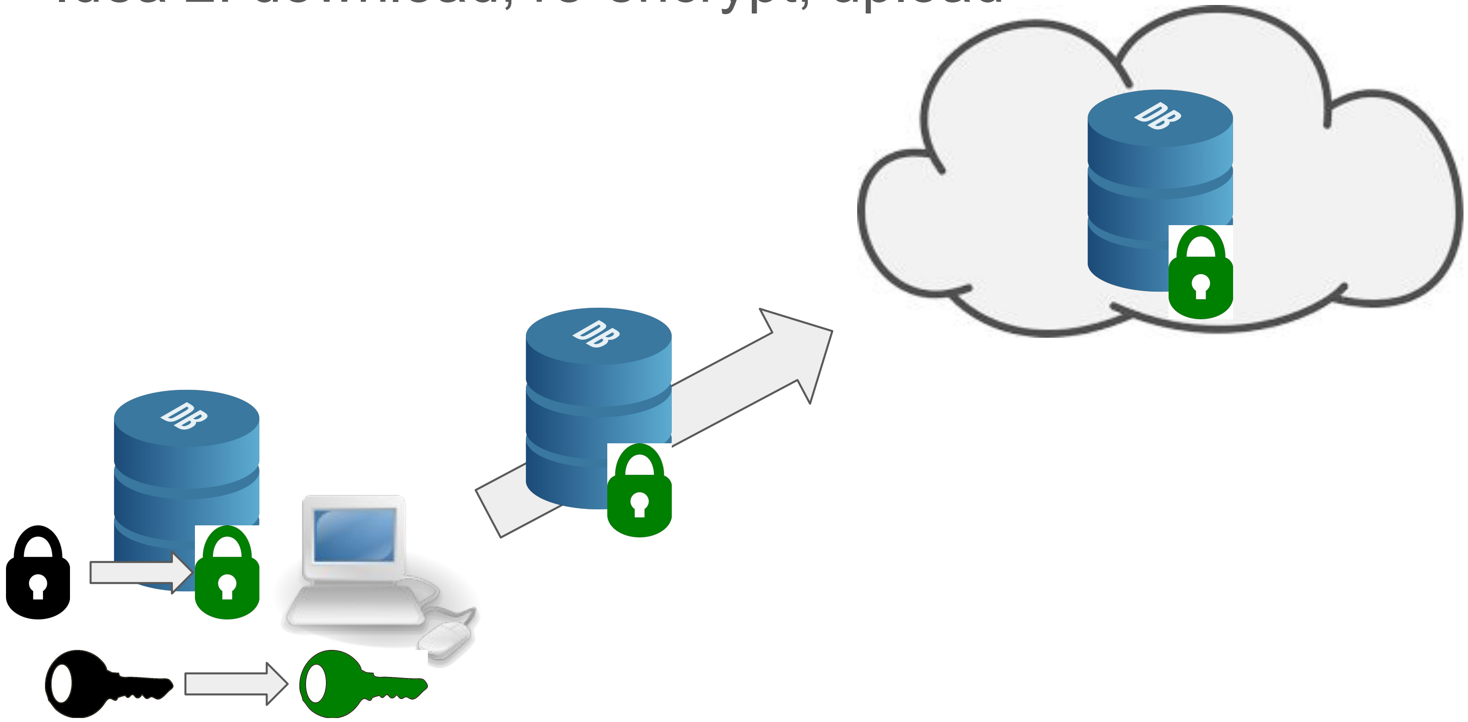
How to Rotate Keys in the Cloud?

Idea 2: download, re-encrypt, upload



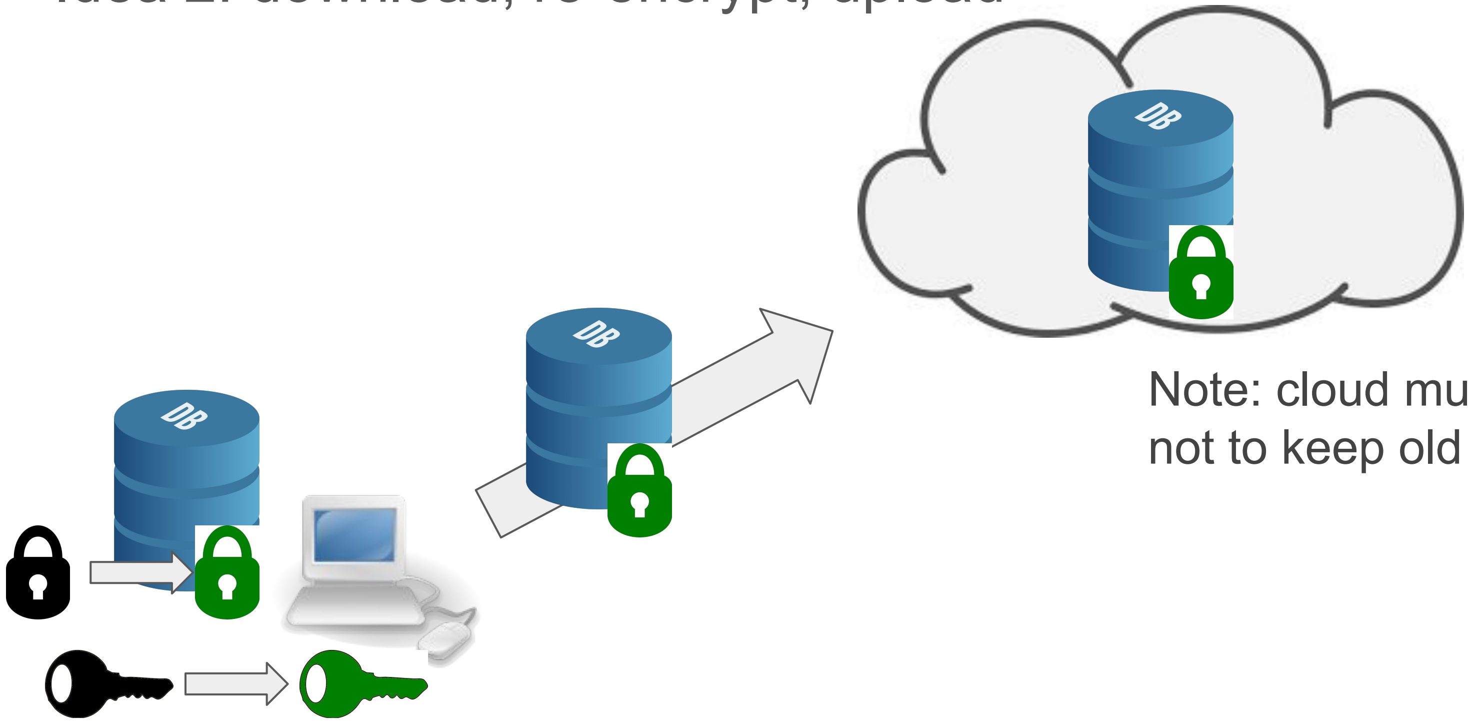
How to Rotate Keys in the Cloud?

Idea 2: download, re-encrypt, upload



How to Rotate Keys in the Cloud?

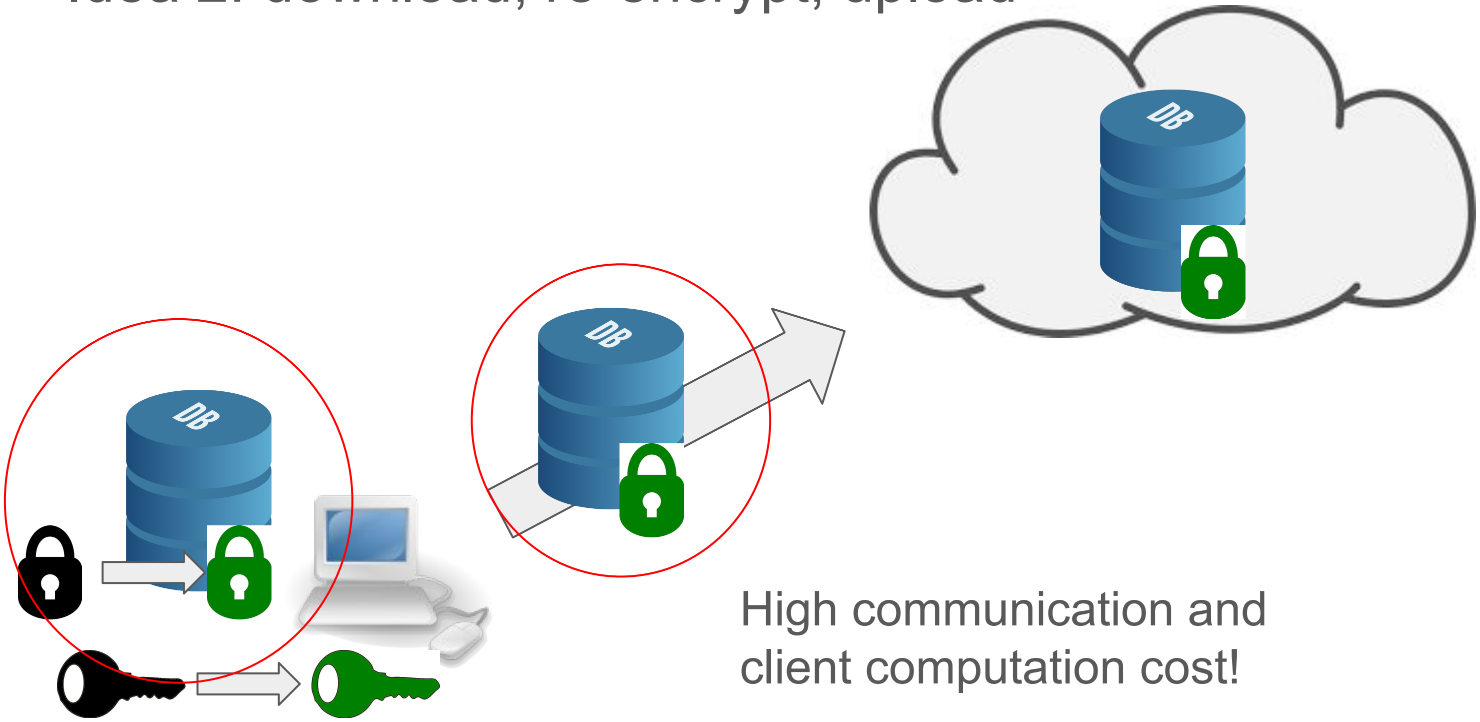
Idea 2: download, re-encrypt, upload



Note: cloud must be trusted not to keep old ciphertexts

How to Rotate Keys in the Cloud?

Idea 2: download, re-encrypt, upload

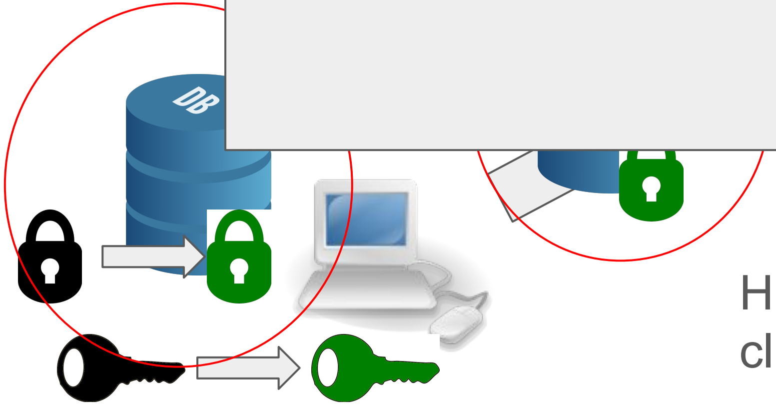


High communication and client computation cost!

How to Rotate Keys in the Cloud?

Idea 2: download, re-encrypt, upload

Can we do better?

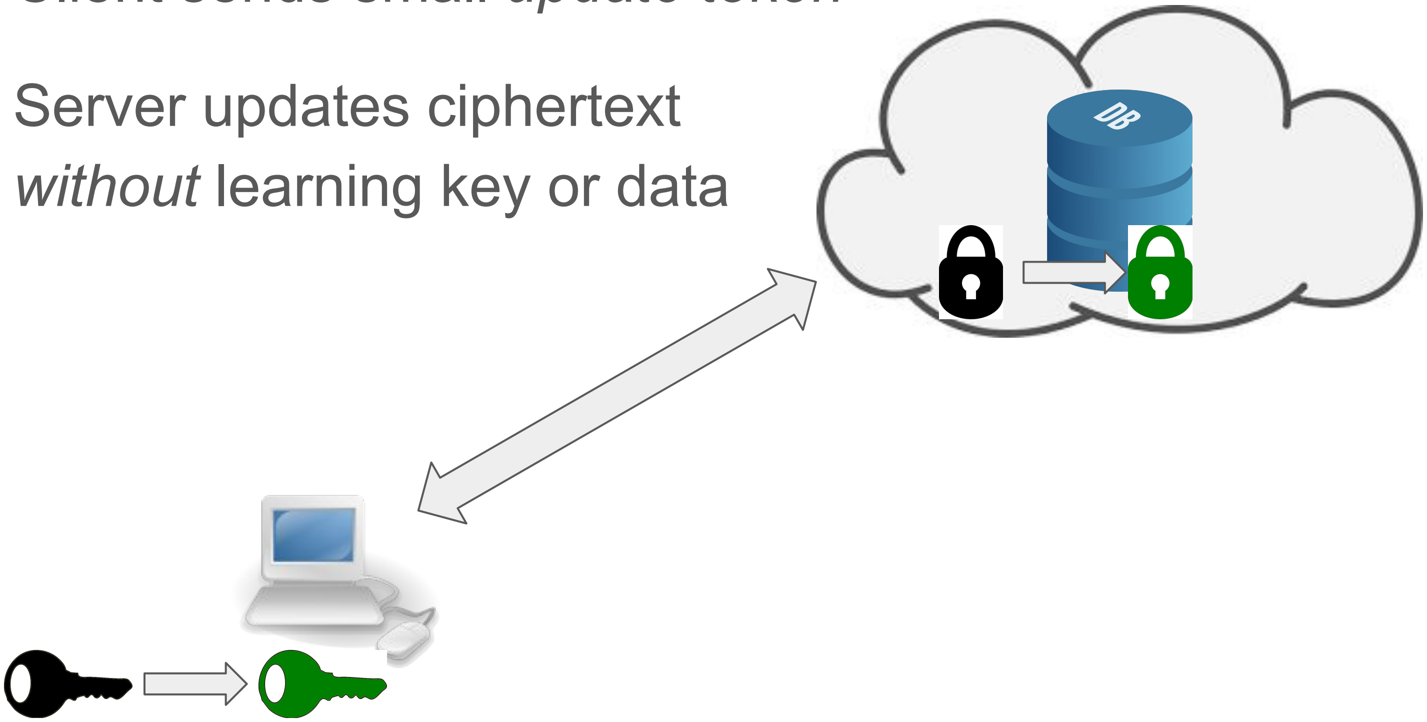


High communication and client computation cost!

Updatable Encryption [BLMR13, EPRS17, LT18, KLR19, BDGJ19]

Client sends small *update token*

Server updates ciphertext
without learning key or data



Our Contributions & Roadmap

Improvements over prior security definitions

- Additional requirements for security

Two new constructions of updatable encryption

- From Nested AES: very fast, only supports *bounded* updates
- From KH-PRF based on RLWE: ~500x faster than prior work

Performance evaluation and comparison to prior work

Recommendations for usage

Security and Functionality Goals

1. Adversary without access to any key does not learn data

Security and Functionality Goals

1. Adversary without access to any key does not learn data
2. Adversary with access to the current key/data cannot get more data than it has already exfiltrated after rekeying

Security and Functionality Goals

1. Adversary without access to any key does not learn data
2. Adversary with access to the current key/data cannot get more data than it has already exfiltrated after rekeying
3. Client-server communication small

Security and Functionality Goals

1. Adversary without access to any key does not learn data
2. Adversary with access to the current key/data cannot get more data than it has already exfiltrated after rekeying
3. Client-server communication small
4. Client computation small

Security and Functionality Goals

1. Adversary without access to any key does not learn data
2. Adversary with access to the current key/data cannot get more data than it has already exfiltrated after rekeying
3. Client-server communication small
4. Client computation small

Limitations

1. Server computation will be linear

Security and Functionality Goals

1. Adversary without access to any key does not learn data
2. Adversary with access to the current key/data cannot get more data than it has already exfiltrated after rekeying
3. Client-server communication small
4. Client computation small

Limitations

1. Server computation will be linear
2. Adversary with ongoing access to key updates will still get data

Defining Security [EPRS17]

Four properties to achieve:

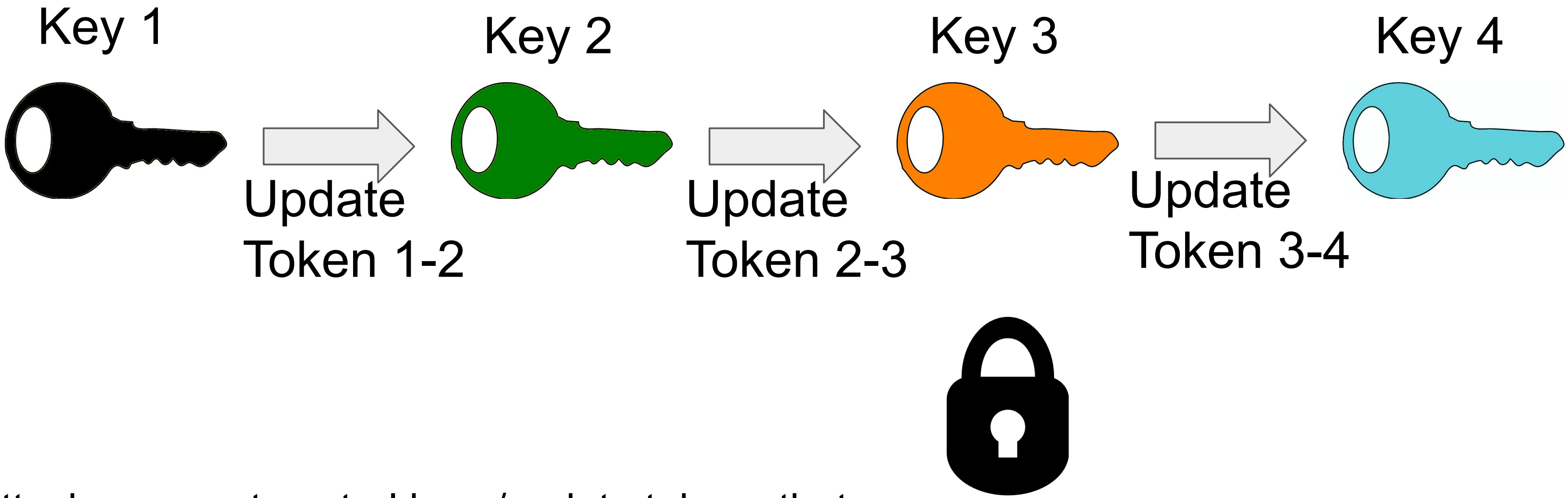
- Correctness
- Compactness
- Confidentiality
- Integrity

Defining Security [EPRS17]

Four properties to achieve:

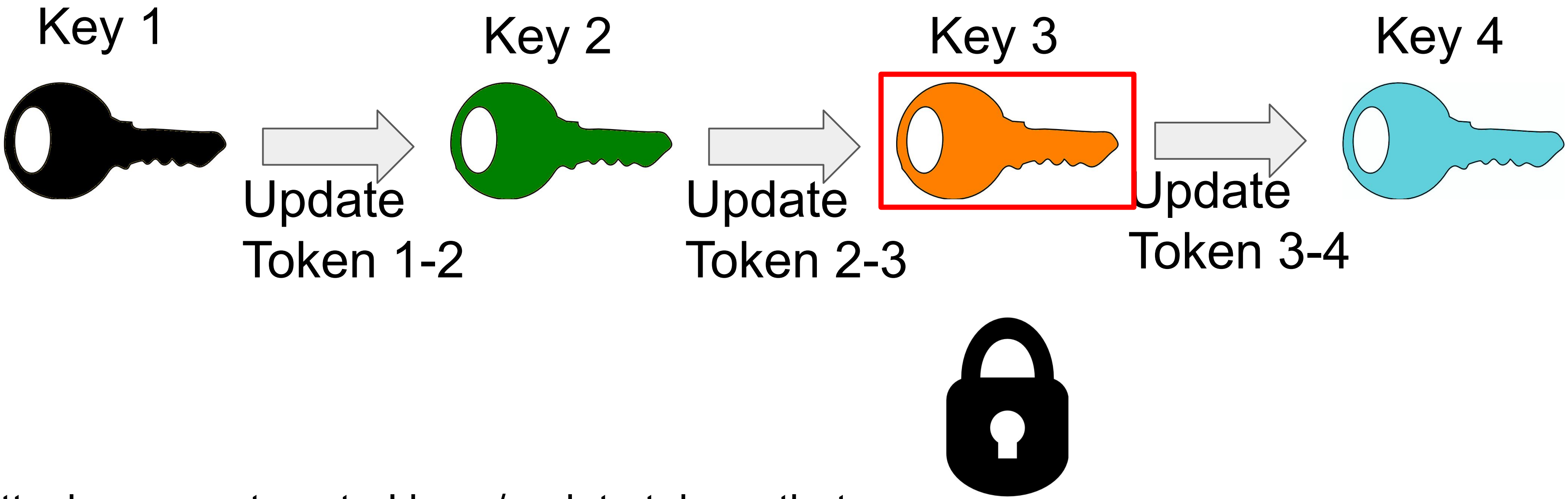
- Correctness
- Compactness
- **Confidentiality**
- Integrity

Confidentiality



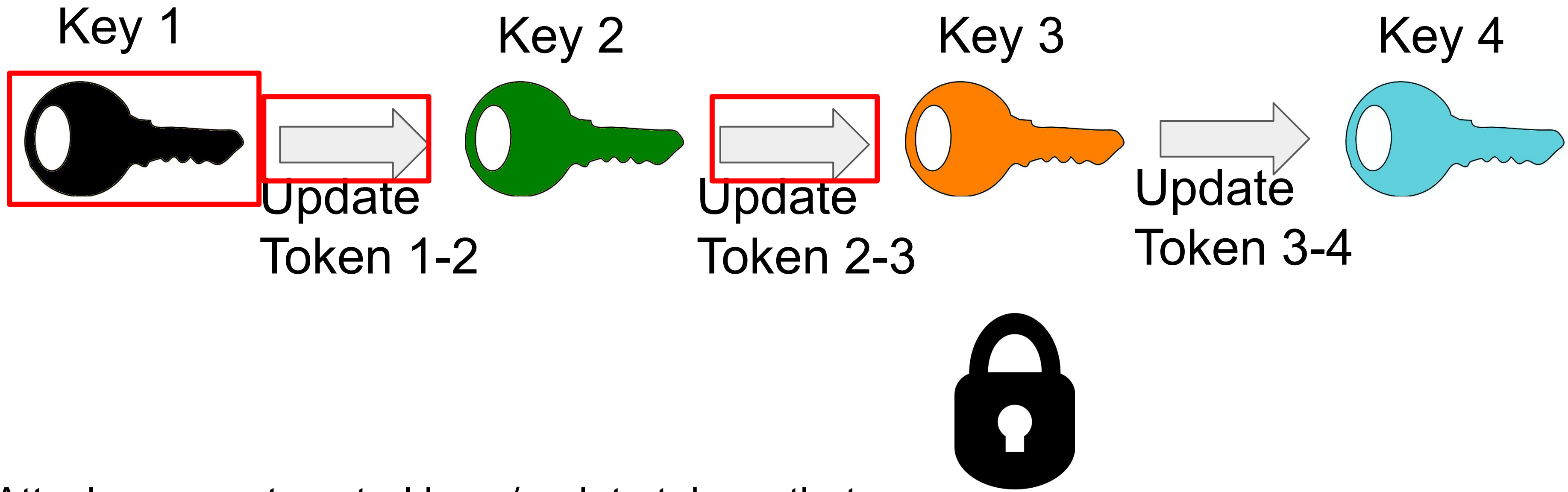
Attacker cannot control keys/update tokens that give a path to key used to encrypt a ciphertext

Confidentiality



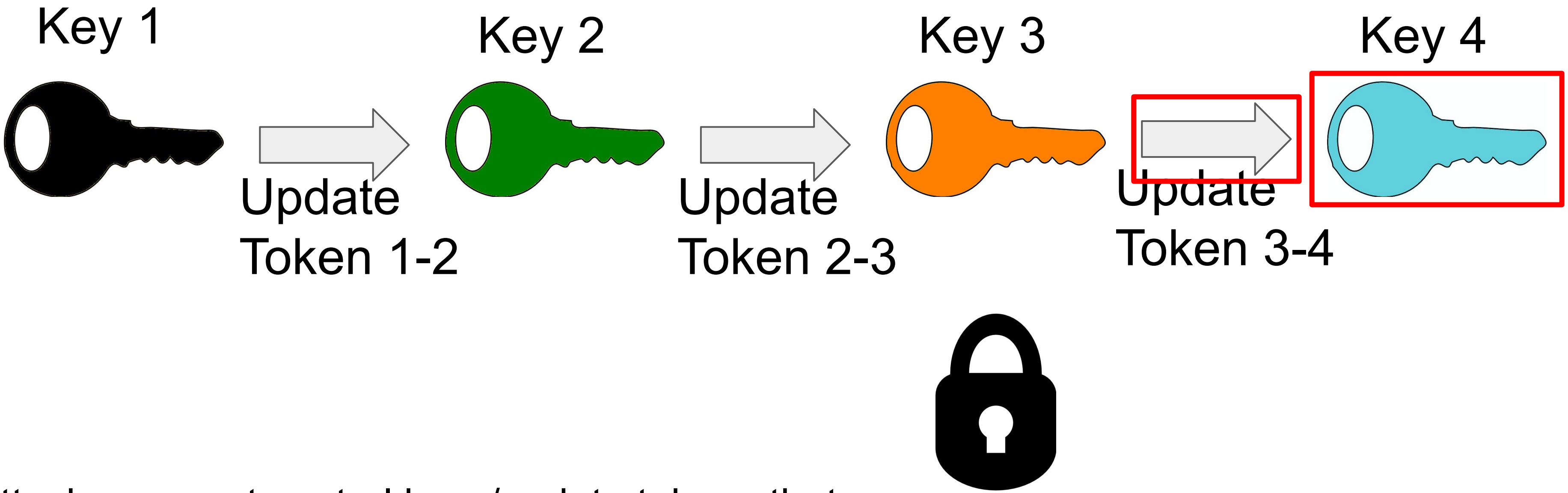
Attacker cannot control keys/update tokens that give a path to key used to encrypt a ciphertext

Confidentiality



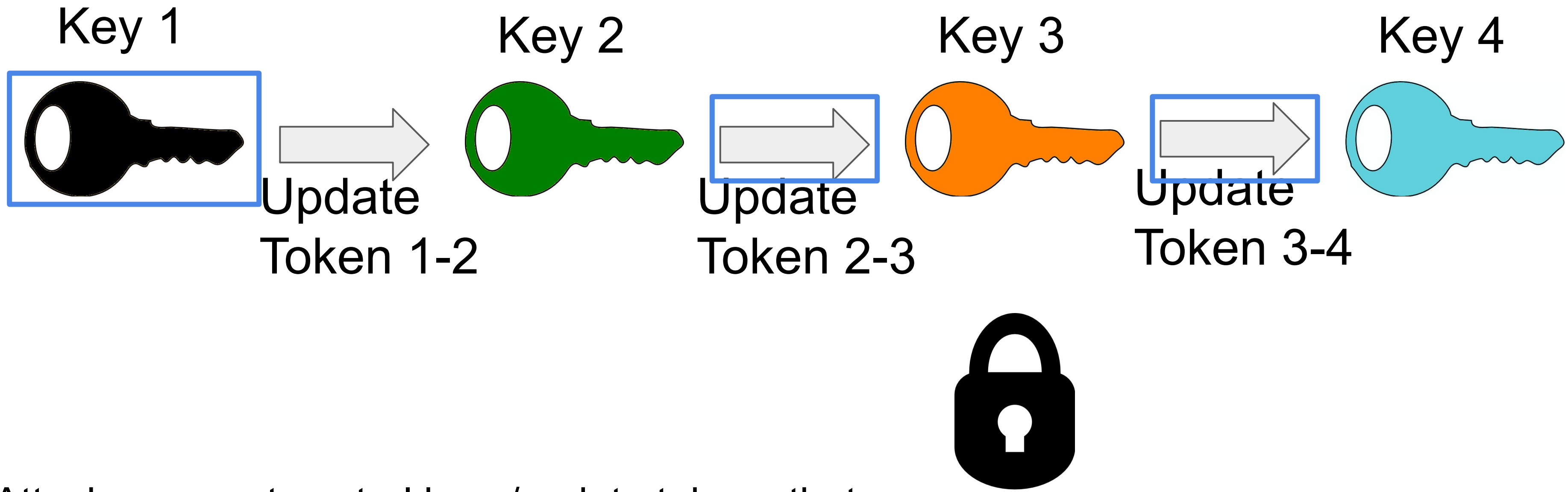
Attacker cannot control keys/update tokens that give a path to key used to encrypt a ciphertext

Confidentiality



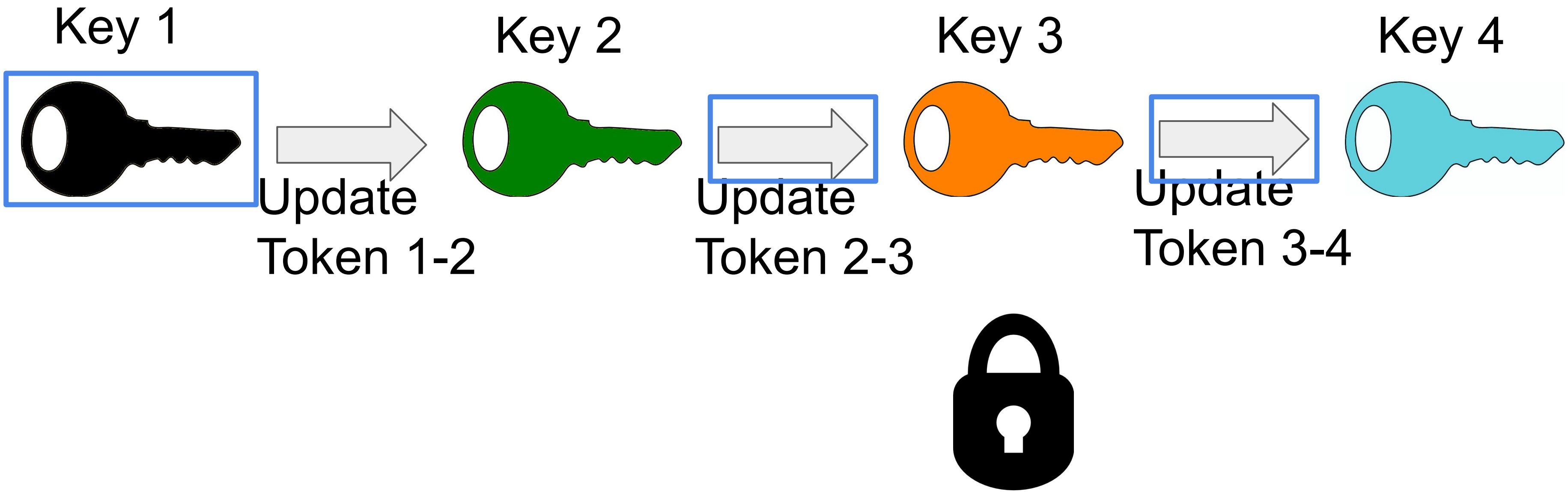
Attacker cannot control keys/update tokens that give a path to key used to encrypt a ciphertext

Confidentiality



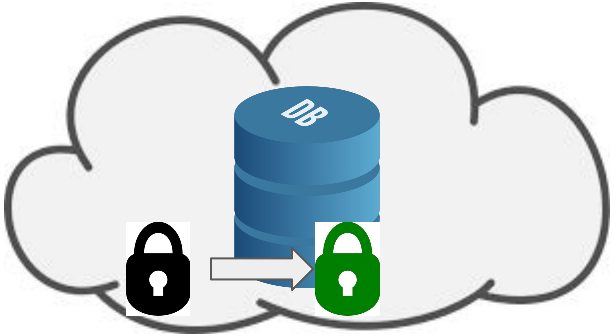
Attacker cannot control keys/update tokens that give a path to key used to encrypt a ciphertext

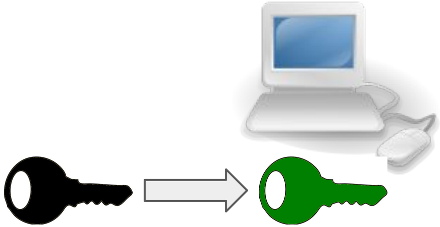
Confidentiality



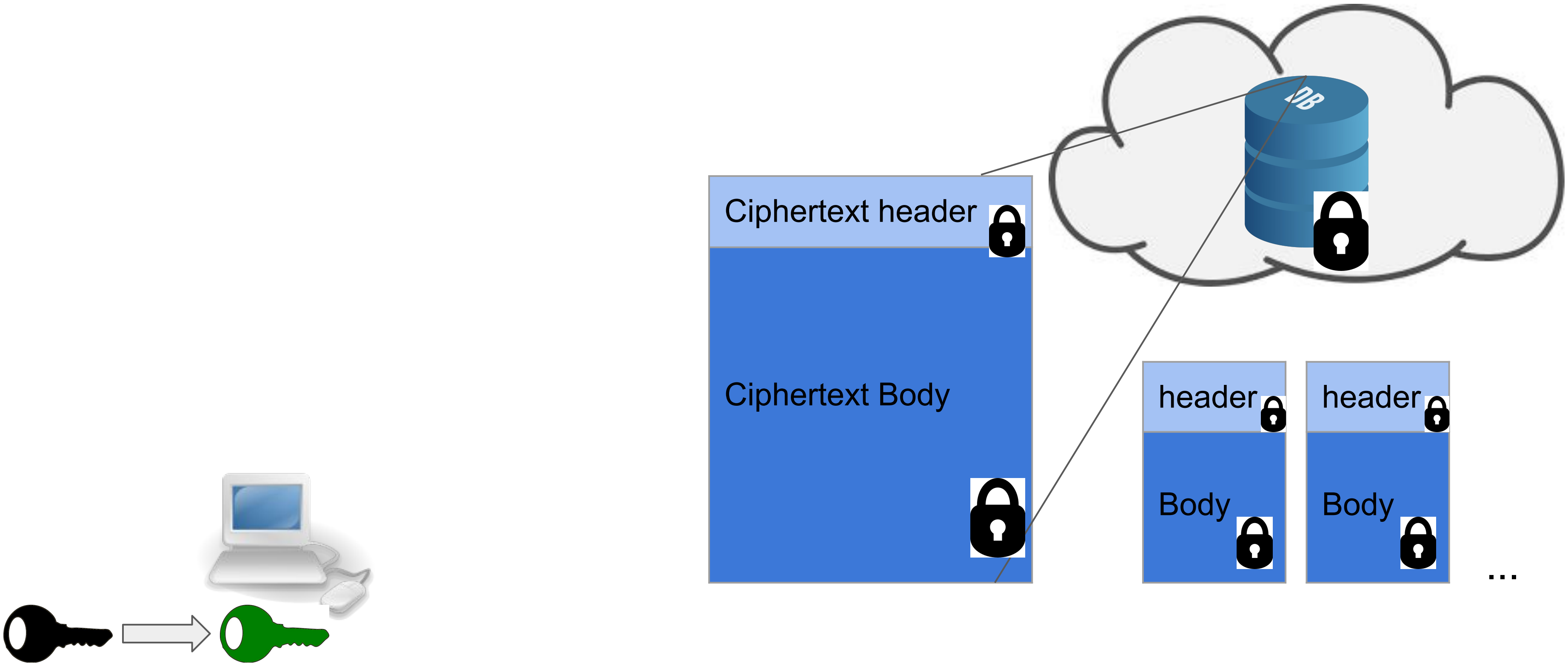
Our definitions additionally require hiding ciphertext age from attacker

Building Updatable Encryption [BLMR13, EPRS17]

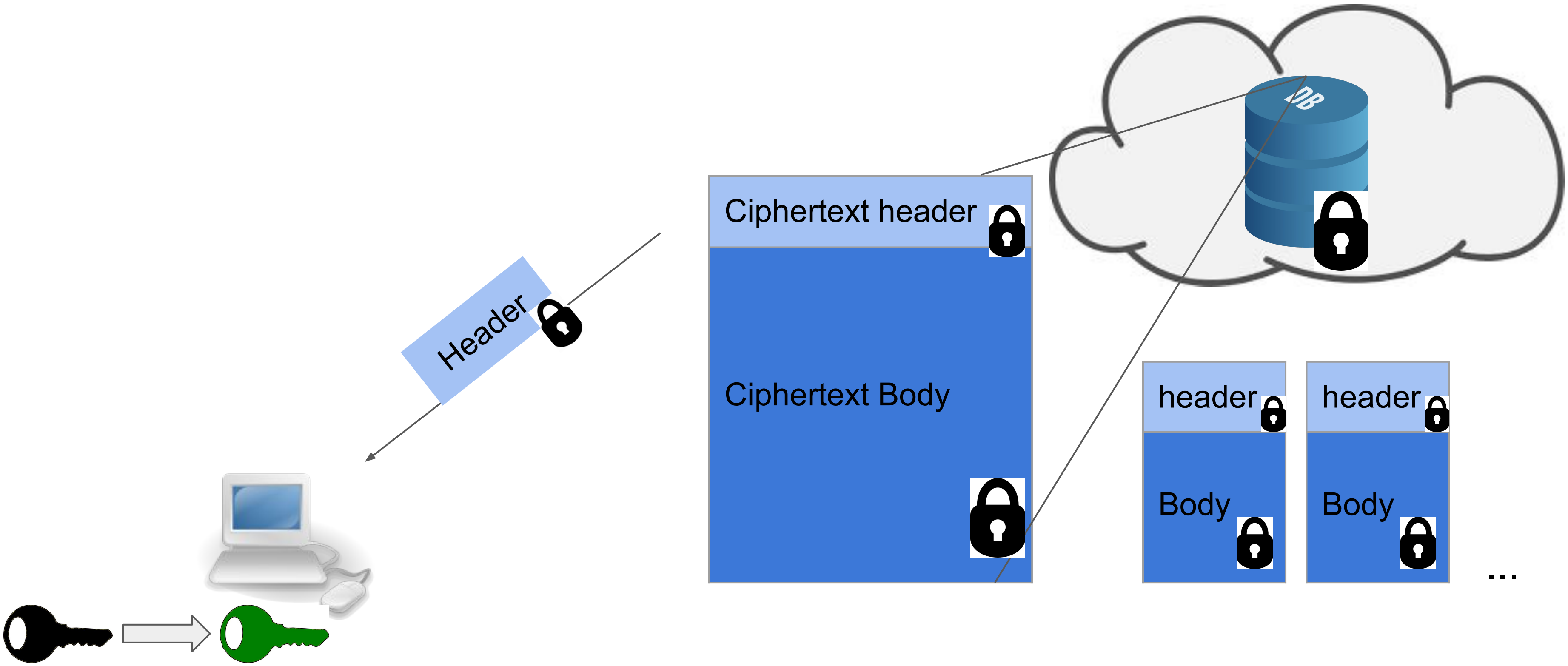




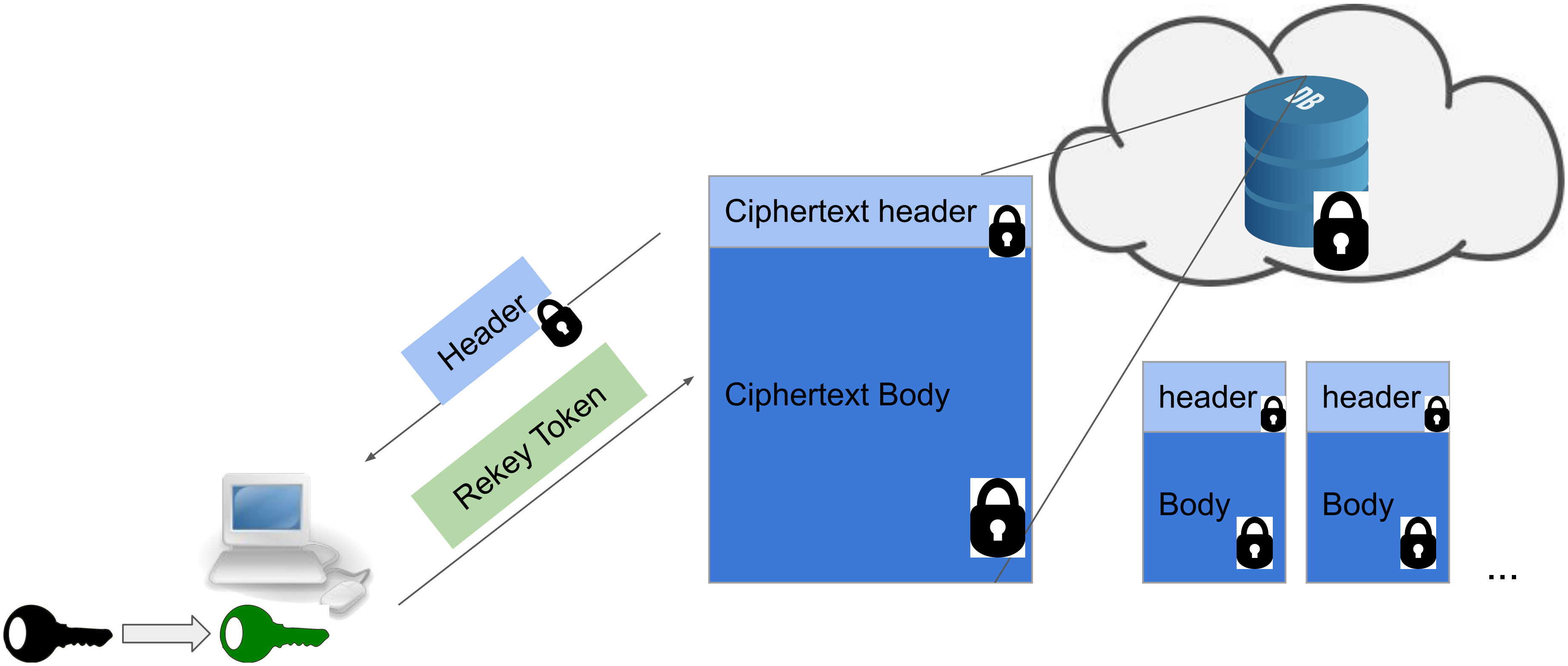
Building Updatable Encryption [BLMR13, EPRS17]



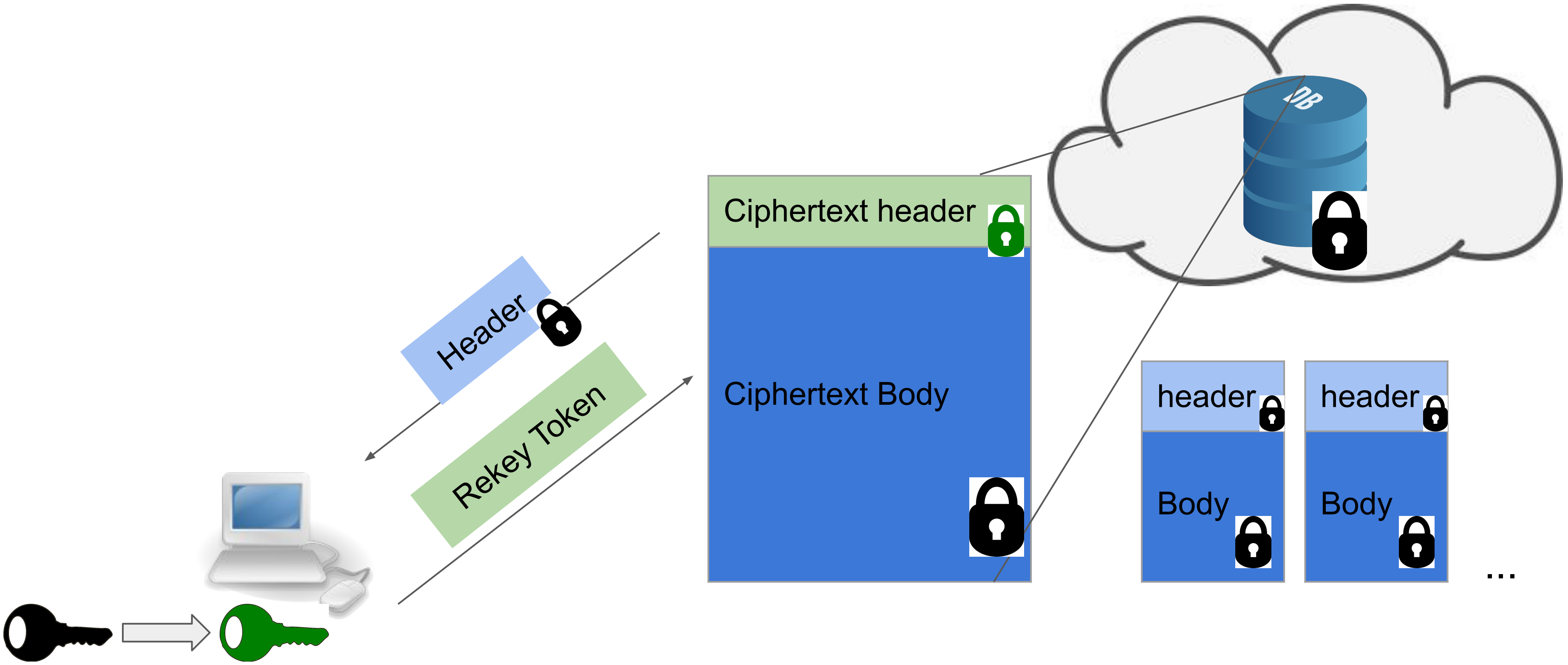
Building Updatable Encryption [BLMR13, EPRS17]



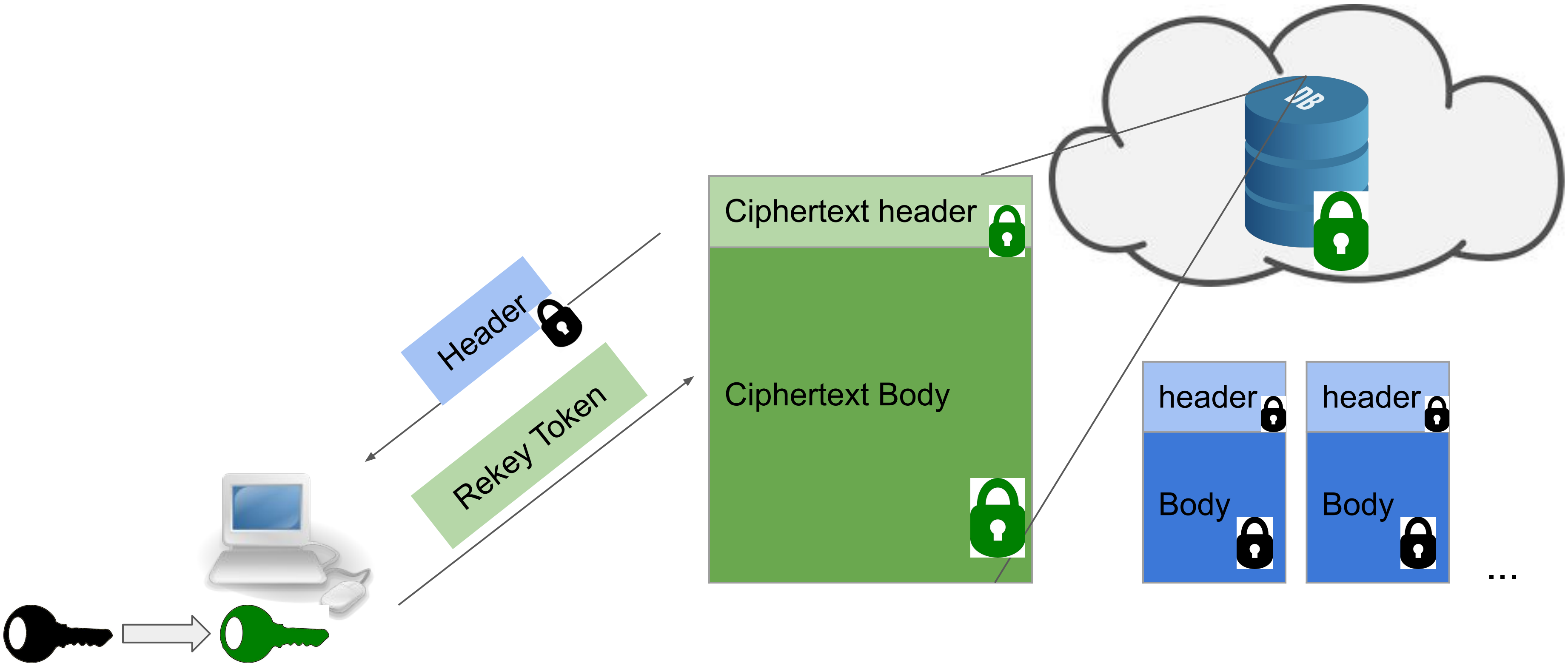
Building Updatable Encryption [BLMR13, EPRS17]



Building Updatable Encryption [BLMR13, EPRS17]

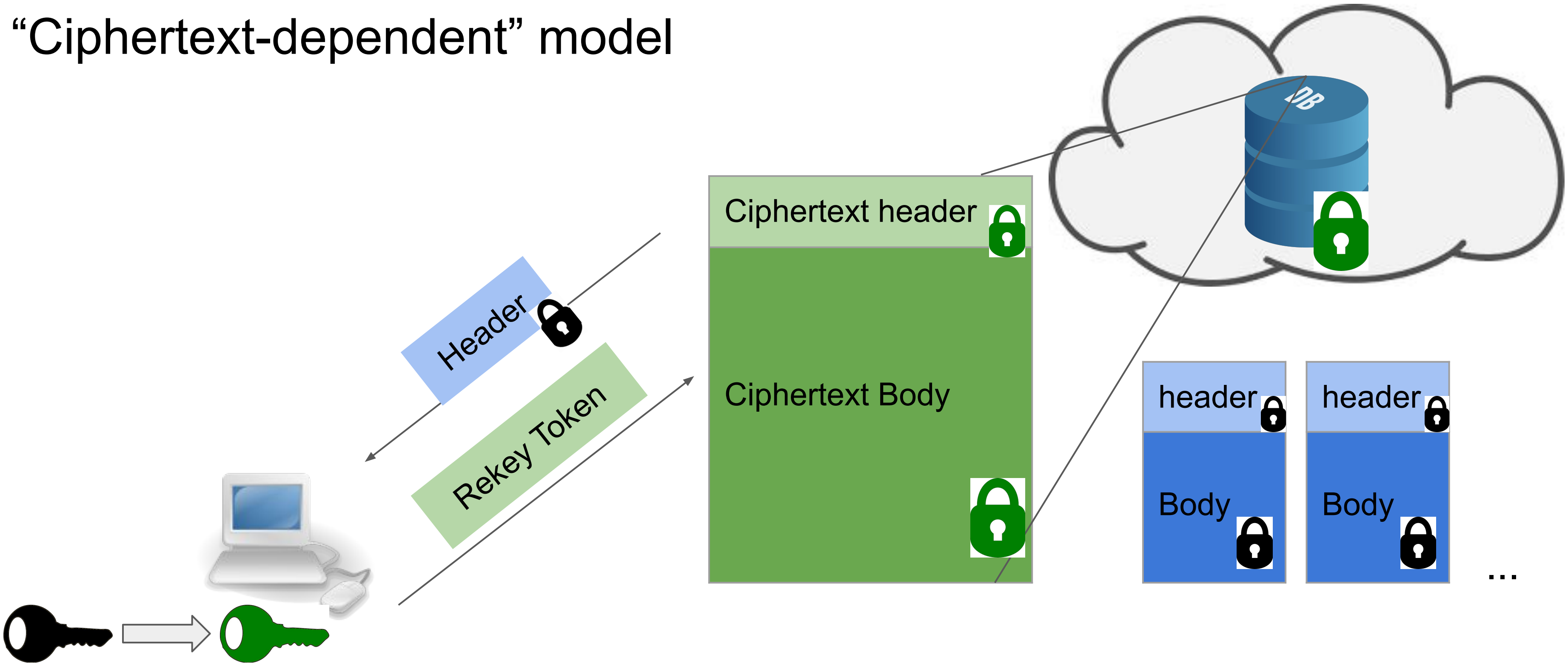


Building Updatable Encryption [BLMR13, EPRS17]



Building Updatable Encryption [BLMR13, EPRS17]

“Ciphertext-dependent” model



Updatable Encryption from Nested AES

Very fast, simple scheme

Only requires authenticated encryption (AES-GCM) and a PRG

Updatable Encryption from Nested AES

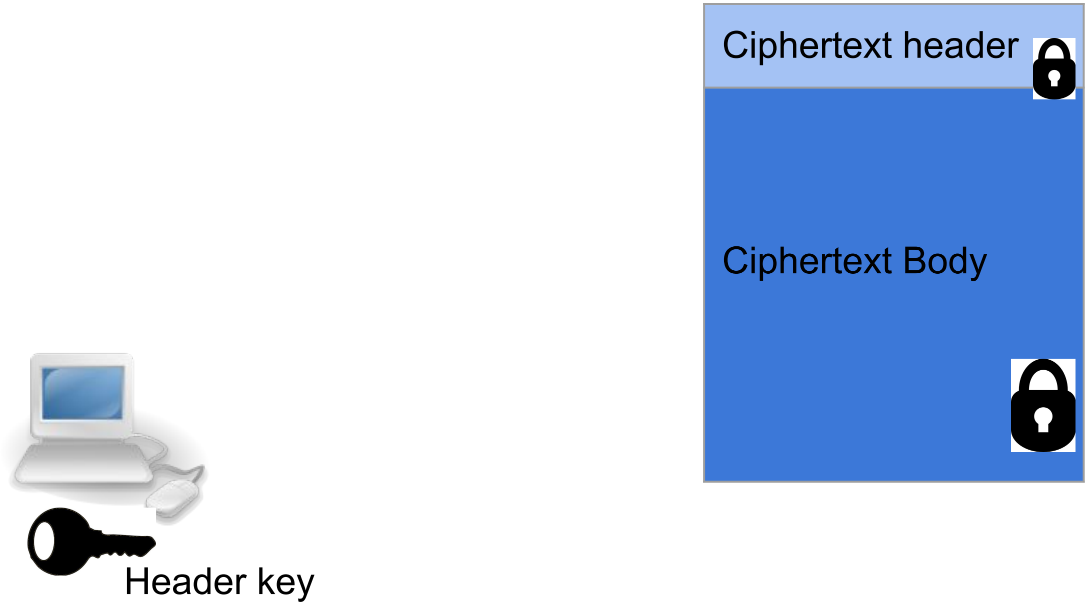
Very fast, simple scheme

Only requires authenticated encryption (AES-GCM) and a PRG

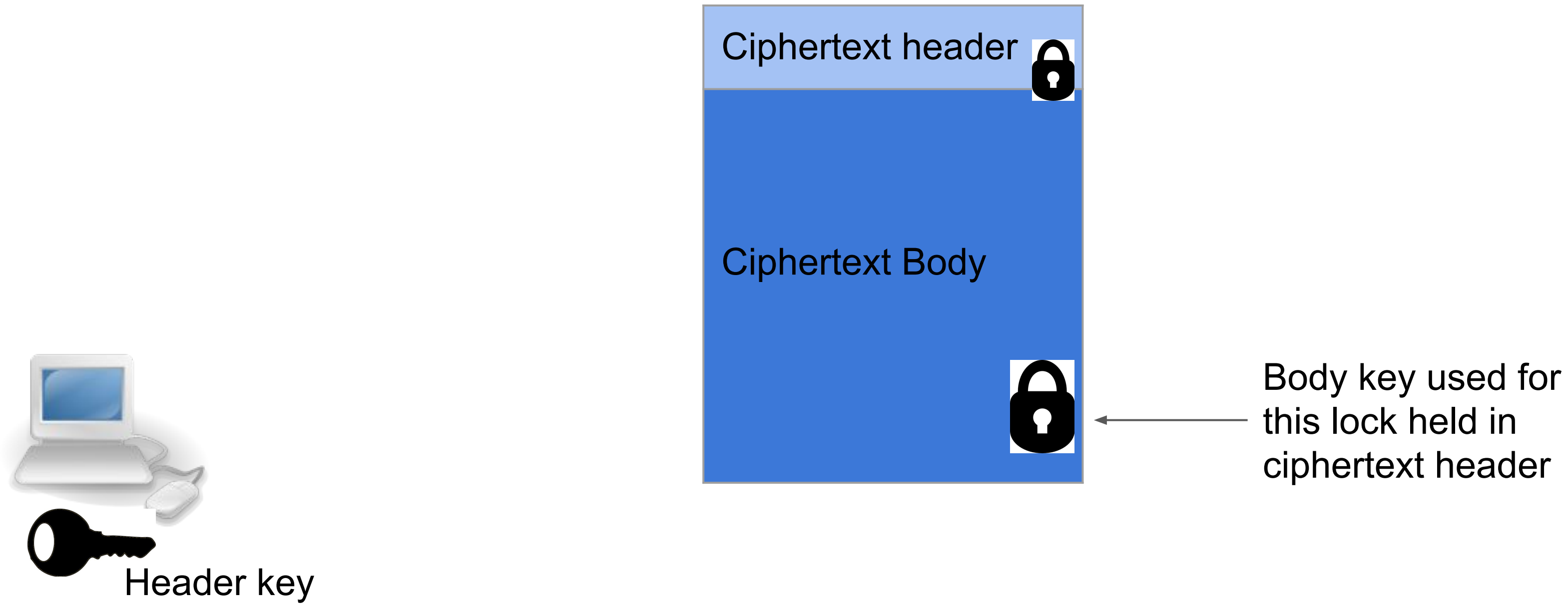
Caveats:

- Only works for a *bounded* number of re-encryptions, decided at encryption time
- Decryption time will be linear in the number of re-encryptions

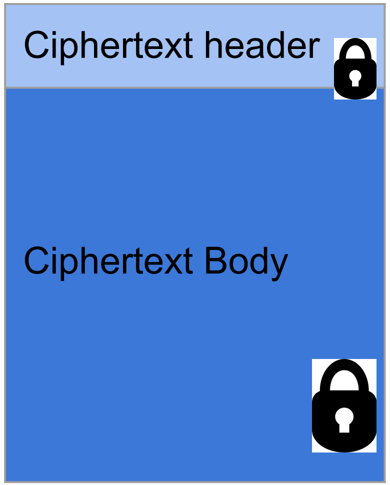
Updatable Encryption from Nested AES

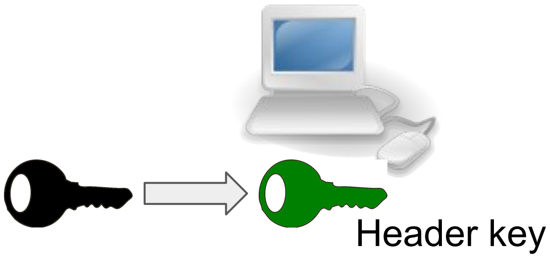


Updatable Encryption from Nested AES

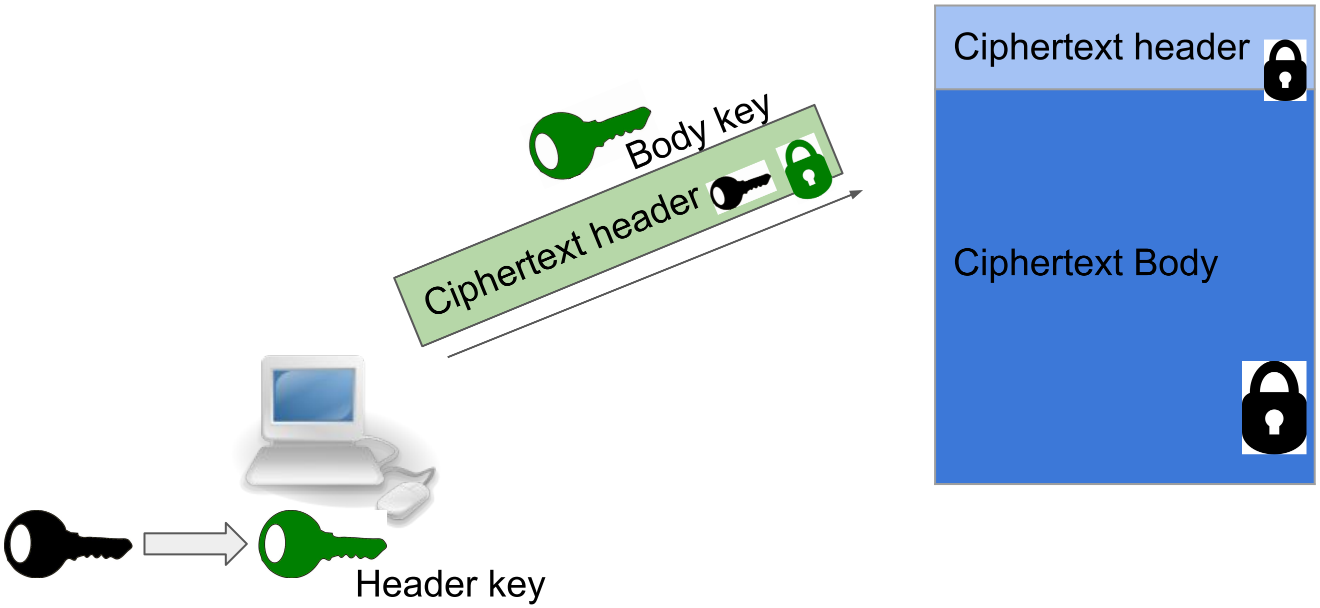


Updatable Encryption from Nested AES

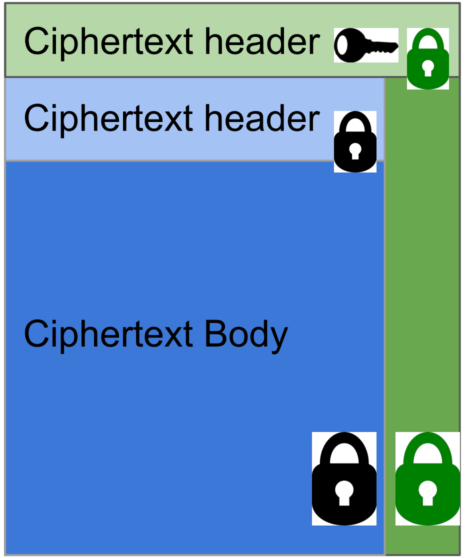


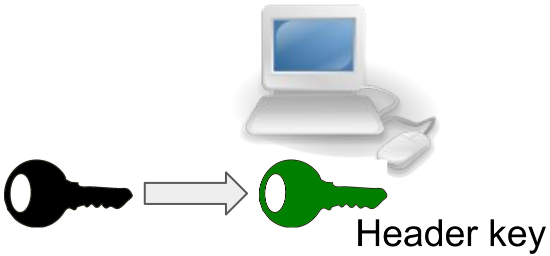


Updatable Encryption from Nested AES

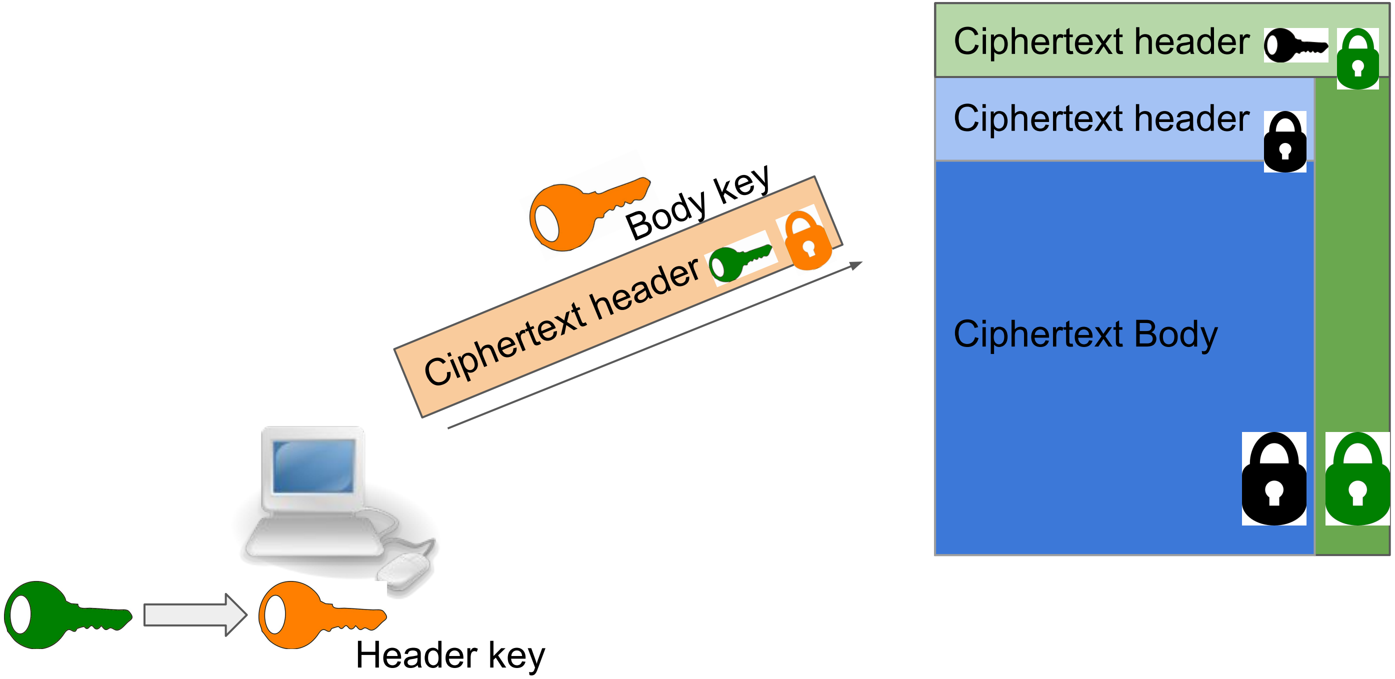


Updatable Encryption from Nested AES

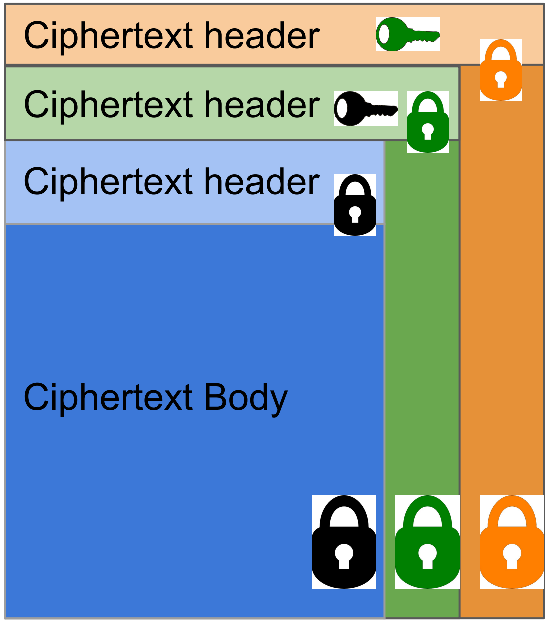


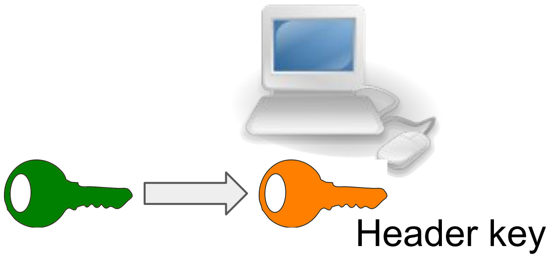


Updatable Encryption from Nested AES



Updatable Encryption from Nested AES

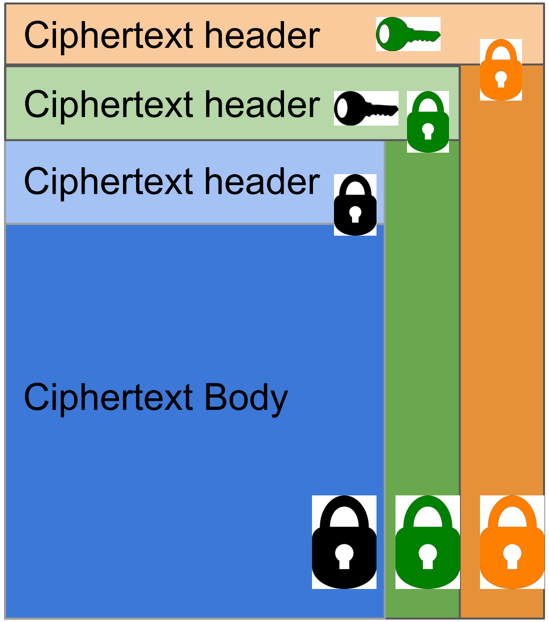




Updatable Encryption from Nested AES

Re-Encryption: wrap previous layer

Decryption: unwrap all layers

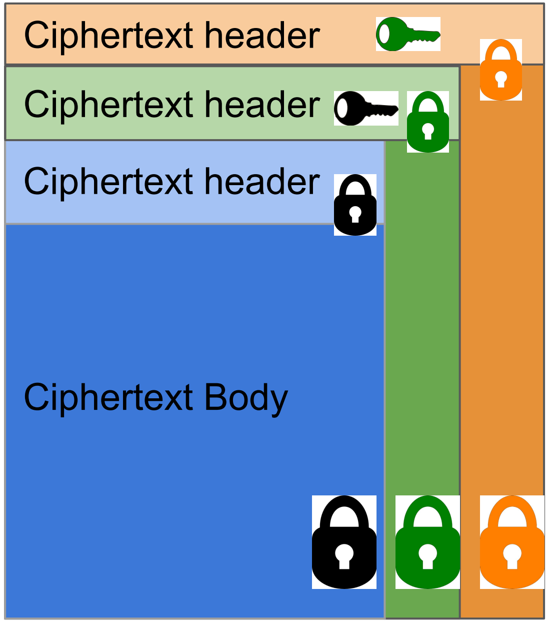


Updatable Encryption from Nested AES

Re-Encryption: wrap previous layer

Decryption: unwrap all layers

Issue: leaks ciphertext age



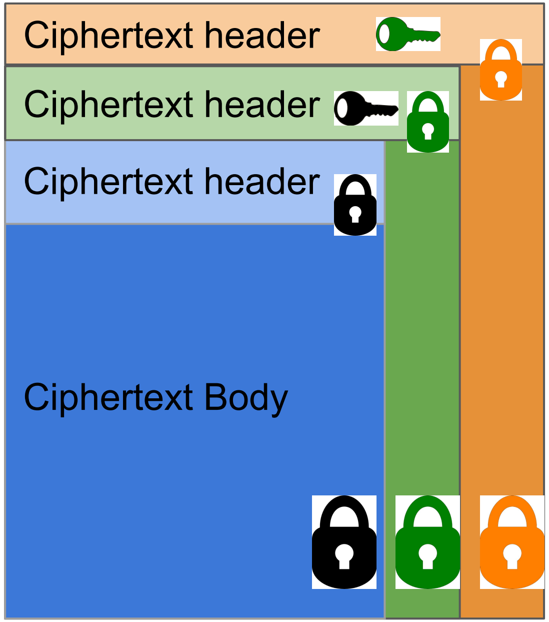
Updatable Encryption from Nested AES

Re-Encryption: wrap previous layer

Decryption: unwrap all layers

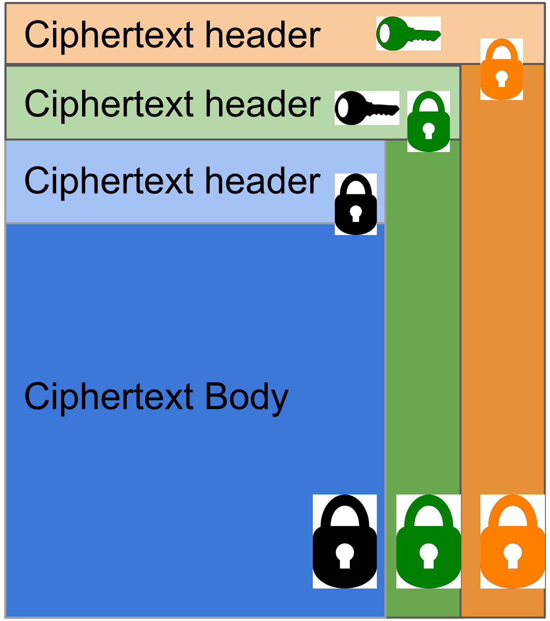
Issue: leaks ciphertext age

Note: this satisfies prior definitions



Updatable Encryption from Nested AES

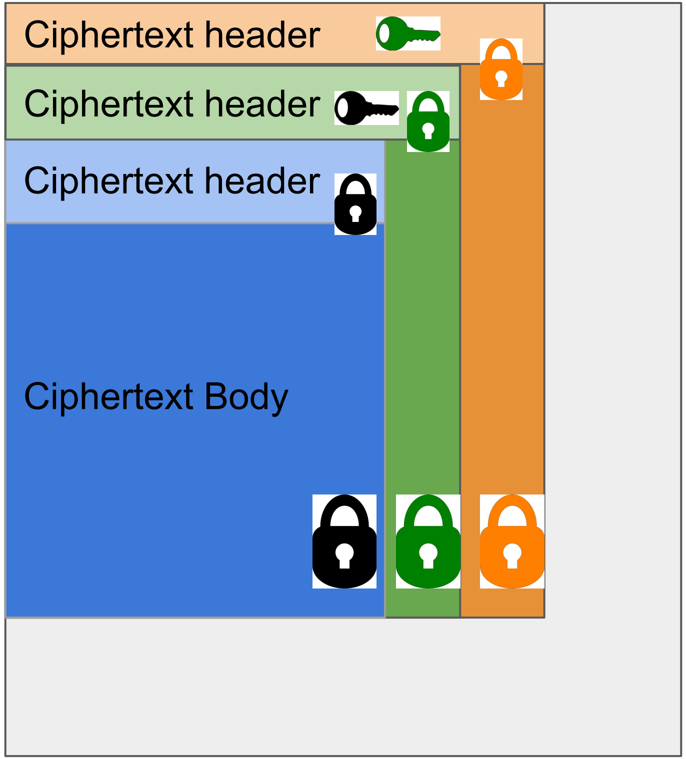
How to hide ciphertext age?



Updatable Encryption from Nested AES

How to hide ciphertext age?

Idea 1: pad up to fixed max size with random data

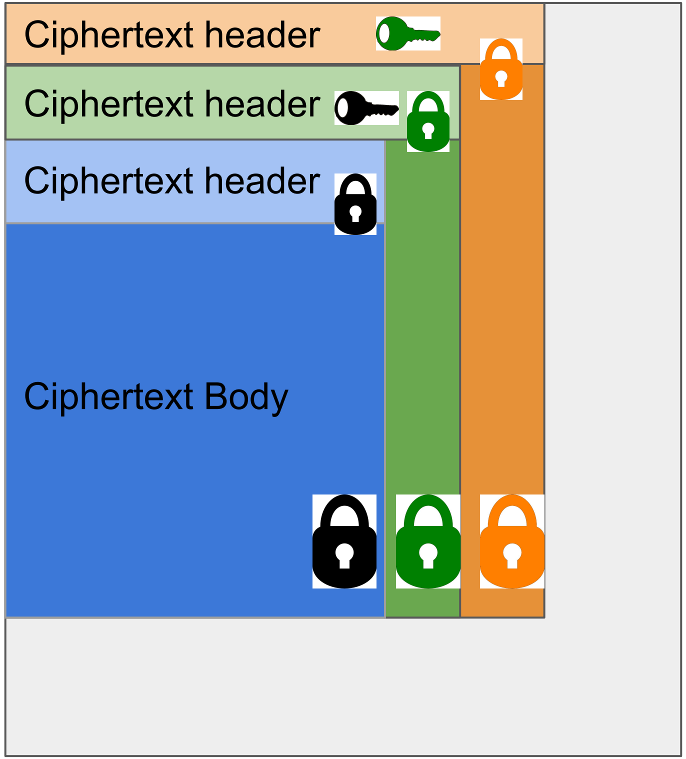


Updatable Encryption from Nested AES

How to hide ciphertext age?

Idea 1: pad up to fixed max size with random data

But this ruins integrity



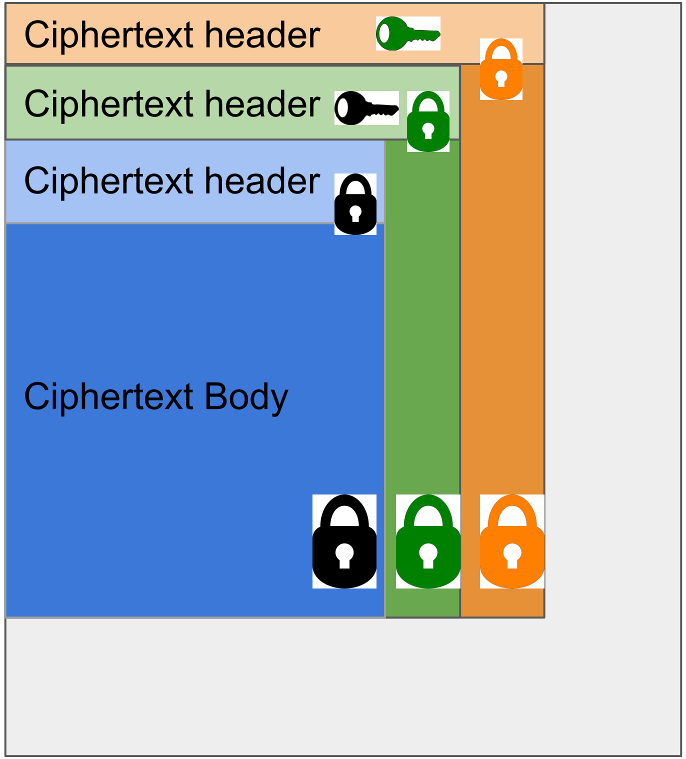
Updatable Encryption from Nested AES

How to hide ciphertext age?

Idea 1: pad up to fixed max size with random data

But this ruins integrity

Idea 2: generate random data from PRG, include seed in header



Updatable Encryption from Nested AES

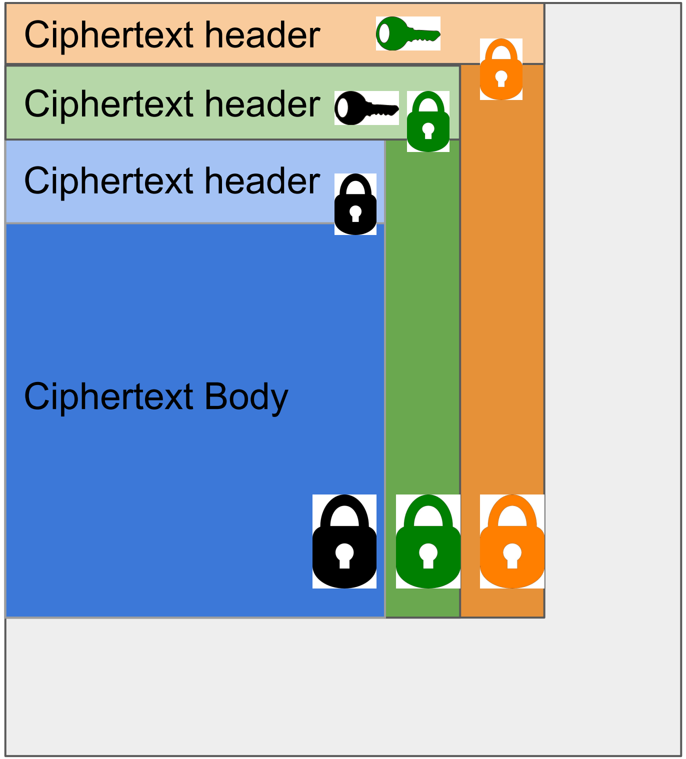
How to hide ciphertext age?

Idea 1: pad up to fixed max size with random data

But this ruins integrity

Idea 2: generate random data from PRG, include seed in header

See paper for full scheme



Updatable Encryption from KH-PRFs [BLMR13, EPRS17]

Supports as many re-encryptions as you want

Decryption time does not depend on number of re-encryptions

Still fast, but slower than nested scheme

New caveat: somewhat weaker integrity and age-hiding guarantee

Tool: Key-Homomorphic PRFs (KHPRFs) [NPR99]

Standard PRF (e.g. AES): *F(k, x)* looks random if not given *k*

Tool: Key-Homomorphic PRFs (KHPRFs) [NPR99]

Standard PRF (e.g. AES): *F(k, x)* looks random if not given *k*

Key-Homomorphic PRF: Same security property, new functionality

Tool: Key-Homomorphic PRFs (KHPRFs) [NPR99]

Standard PRF (e.g. AES): *F(k, x)* looks random if not given *k*

Key-Homomorphic PRF: Same security property, new functionality

$$F(k_1, x) \boxplus F(k_2, x) = F(k_1 + k_2, x)$$

Tool: Key-Homomorphic PRFs (KHPRFs) [NPR99]

Standard PRF (e.g. AES): *F(k, x)* looks random if not given *k*

Key-Homomorphic PRF: Same security property, new functionality

$$F(k_1, x) \boxplus F(k_2, x) = F(k_1 + k_2, x)$$

Example: $F(k,x) = H(x)^k$

Tool: Key-Homomorphic PRFs (KHPRFs) [NPR99]

Standard PRF (e.g. AES): *F(k, x)* looks random if not given *k*

Key-Homomorphic PRF: Same security property, new functionality

$$F(k_1, x) \boxplus F(k_2, x) = F(k_1 + k_2, x)$$

Example: $F(k,x) = H(x)^k$

$$F(k_1, x) * F(k_2, x) = H(x)^{k1} * H(x)^{k2} = H(x)^{k1+k2} = F(k_1 + k_2, x)$$

Updatable Encryption from KH-PRFs [EPRS17]

Ciphertext header:

Authenticated Encryption of *H(msg)* and KH-PRF key k_1

Updatable Encryption from KH-PRFs [EPRS17]

Ciphertext header:

Authenticated Encryption of *H(msg)* and KH-PRF key k_1

Ciphertext body:

Encryption of msg in counter mode using KH-PRF

Updatable Encryption from KH-PRFs [EPRS17]

Ciphertext header:

Authenticated Encryption of *H(msg)* and KH-PRF key k_1

Ciphertext body:

Encryption of msg in counter mode using KH-PRF

$c_0 = m_0 + F(k_1, 0)$
$c_1 = m_1 + F(k_1, 1)$
…
$c_n = m_n + F(k_1, n)$

Updatable Encryption from KH-PRFs [EPRS17]

Ciphertext header:

Authenticated Encryption of *H(msg)* and KH-PRF key k_1

Ciphertext body:

Encryption of msg in counter mode using KH-PRF

$c_0 = m_0 + F(k_1, 0)$
$c_1 = m_1 + F(k_1, 1)$
…
$c_n = m_n + F(k_1, n)$

Update process:

1. Download/decrypt header
2. Pick key k_2
3. Upload new header and $k_{up} = k_2 - k_1$

Server updates body encryptions with k_{up}

Updatable Encryption from KH-PRFs [EPRS17]

<u>Ciphertext header:</u>

Authenticated Encryption of *H(msg)* and KH-PRF key k_1

<u>Ciphertext body:</u>

Encryption of msg in counter mode using KH-PRF

$c_0' = c_0 + F(k_{up}, 0)$
$c_1' = c_1 + F(k_{up}, 1)$
…
$c_n' = c_n + F(k_{up}, n)$

<u>Update process:</u>

1. Download/decrypt header
2. Pick key k_2
3. Upload new header <u>and</u> $k_{up} = k_2 - k_1$

Server updates body encryptions with k_{up}

Updatable Encryption from KH-PRFs [EPRS17]

Ciphertext header:

Authenticated Encryption of *H(msg)* and KH-PRF key k_1

Ciphertext body:

Encryption of msg in counter mode using KH-PRF

$c_0' = c_0 + F(k_{up}, 0) = m_0 + F(k_2, 0)$
$c_1' = c_1 + F(k_{up}, 1) = m_1 + F(k_2, 1)$
…
$c_n' = c_n + F(k_{up}, n) = m_n + F(k_2, n)$

Update process:

1. Download/decrypt header
2. Pick key k_2
3. Upload new header and $k_{up} = k_2 - k_1$

Server updates body encryptions with k_{up}

Almost KH-PRFs [BLMR13]

EPRS17 uses a KH-PRF based on the DDH assumption*

$$F(k_1, x) + F(k_2, x) = F(k_1 + k_2, x)$$

*In Random Oracle model

Almost KH-PRFs [BLMR13]

EPRS17 uses a KH-PRF based on the DDH assumption*

$$F(k_1, x) + F(k_2, x) = F(k_1 + k_2, x)$$

We use a new *almost KH-PRF* based on the Ring-LWE assumption*

*In Random Oracle model

Almost KH-PRFs [BLMR13]

EPRS17 uses a KH-PRF based on the DDH assumption*

$$F(k_1, x) + F(k_2, x) = F(k_1 + k_2, x)$$

We use a new *almost KH-PRF* based on the Ring-LWE assumption*

$$F(k_1, x) + F(k_2, x) = F(k_1 + k_2, x) \underline{+ e} \qquad \text{(where } e \text{ is small in } Z_q^{\,n}\text{)}$$

*In Random Oracle model

Almost KH-PRFs [BLMR13]

EPRS17 uses a KH-PRF based on the DDH assumption*

$$F(k_1, x) + F(k_2, x) = F(k_1 + k_2, x)$$

We use a new *almost KH-PRF* based on the Ring-LWE assumption*

$$F(k_1, x) + F(k_2, x) = F(k_1 + k_2, x) \underline{+ e} \qquad \text{(where } e \text{ is small in } Z_q^n\text{)}$$

See paper for construction

*In Random Oracle model

Almost KH-PRFs [BLMR13]

EPRS17 uses a KH-PRF based on the DDH assumption*

$$F(k_1, x) + F(k_2, x) = F(k_1 + k_2, x)$$

We use a new *almost KH-PRF* based on the Ring-LWE assumption*

$$F(k_1, x) + F(k_2, x) = F(k_1 + k_2, x) \underline{+\, e} \qquad \text{(where } e \text{ is small in } Z_q^n)$$

See paper for construction

Result: ~500x faster performance

*In Random Oracle model

Almost KH-PRFs [BLMR13]

EPRS17 uses a KH-PRF based on the DDH assumption*

$$F(k_1, x) + F(k_2, x) = F(k_1 + k_2, x)$$

We use a new *almost KH-PRF* based on the Ring-LWE assumption*

$$F(k_1, x) + F(k_2, x) = F(k_1 + k_2, x) \underline{+ e} \qquad \text{(where } e \text{ is small in } Z_q^n\text{)}$$

See paper for construction

Result: ~500x faster performance …but how to handle the noise?

*In Random Oracle model

Updatable Encryption from *Almost* KH-PRFs

$F(k_1, x) + F(k_2, x) = F(k_1 + k_2, x) \underline{+ e}$ (where e is small)

Issue: noisy KH-PRF corrupts message

Updatable Encryption from *Almost* KH-PRFs

$F(k_1, x) + F(k_2, x) = F(k_1 + k_2, x) \underline{+ e}$ (where e is small)

Issue: noisy KH-PRF corrupts message

General solution: error correcting codes

Updatable Encryption from *Almost* KH-PRFs

$$F(k_1, x) + F(k_2, x) = F(k_1 + k_2, x) \underline{+ e} \qquad \text{(where } e \text{ is small)}$$

Issue: noisy KH-PRF corrupts message

General solution: error correcting codes

Observation: noise is always on low-order bits

Updatable Encryption from *Almost* KH-PRFs

$F(k_1, x) + F(k_2, x) = F(k_1 + k_2, x) \underline{+ e}$ (where e is small)

Issue: noisy KH-PRF corrupts message

General solution: error correcting codes

Observation: noise is always on low-order bits

Simple solution: pad low-order bits of each block with zeros

Evaluation

Encryption and Re-encryption

Throughput for encrypting/re-encrypting 32KB messages (MB/sec)

	ReCrypt [EPRS17]	Almost KH-PRF	Nested (128 layers)
Encrypt	0.12	61.90	1836.9
Re-encrypt	0.15	83.06	2606.8

Encryption and Re-encryption

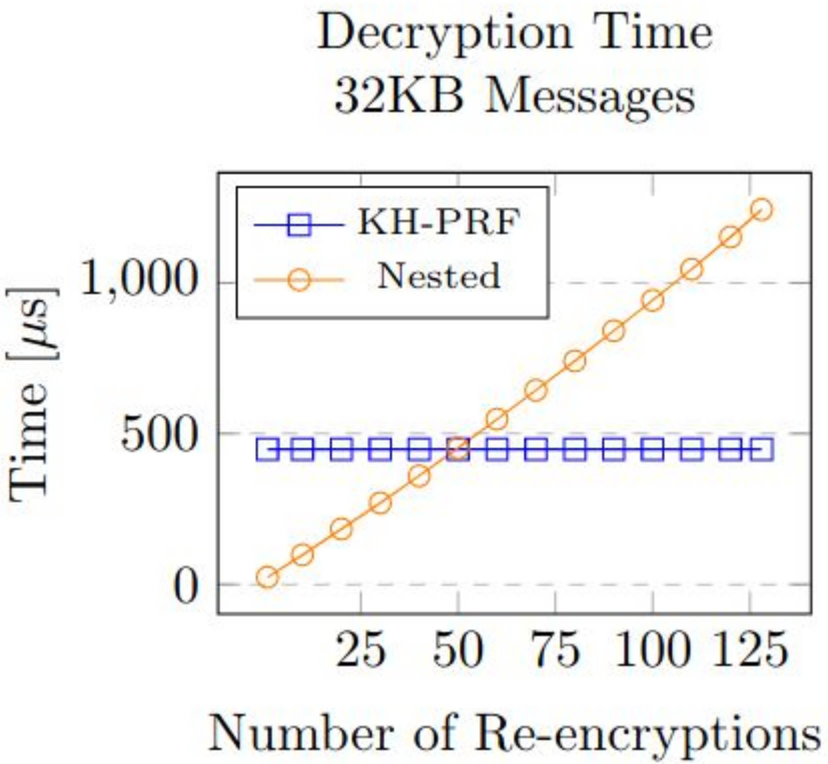
Throughput for encrypting/re-encrypting 32KB messages (MB/sec)

	ReCrypt [EPRS17]	Almost KH-PRF	Nested (128 layers)
Encrypt	0.12	61.90	1836.9
Re-encrypt	0.15	83.06	2606.8

Almost KH-PRF is ~500x faster than ReCrypt

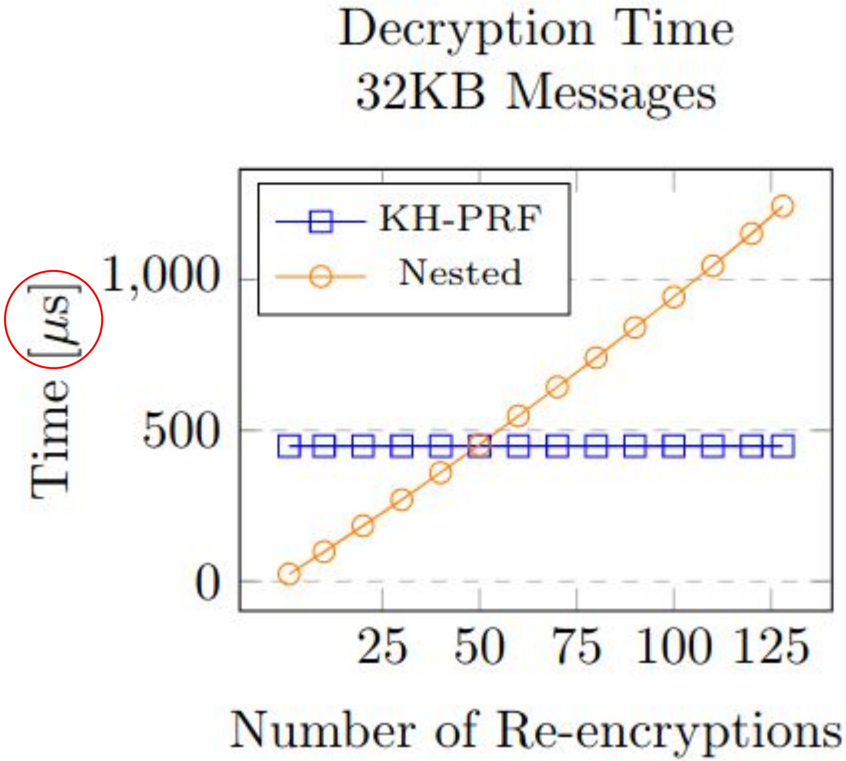
Nested AES is ~30x faster than almost KH-PRF

Decryption



Decryption Time
32KB Messages
KH-PRF
Nested
Time [μs]
1,000
500
0
25
50
75
100
125
Number of Re-encryptions

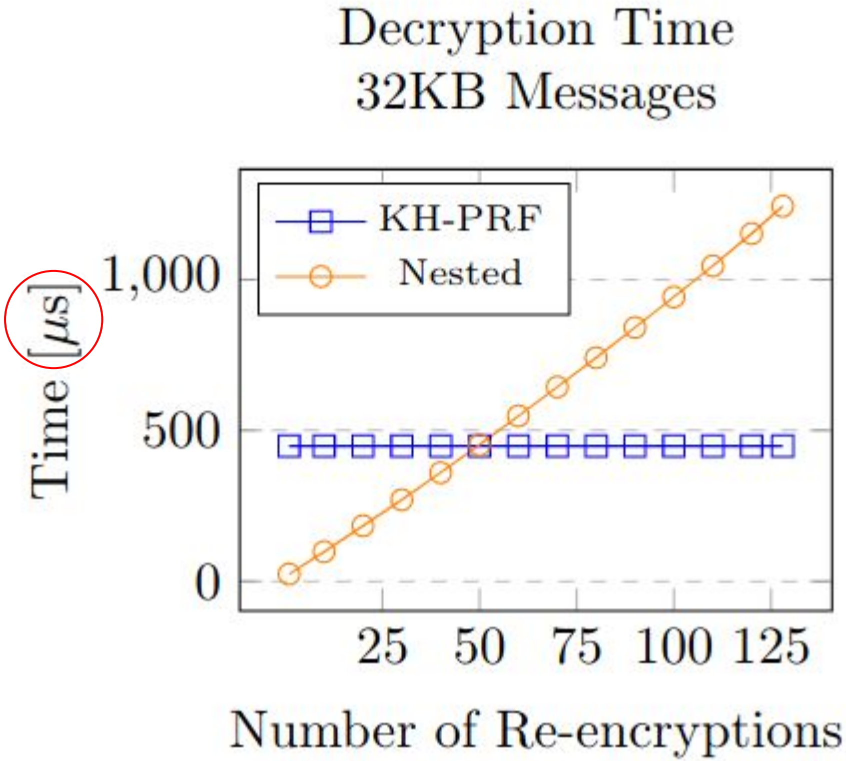
Decryption


Decryption Time
32KB Messages
KH-PRF
Nested
Time [μs]
1,000
500
0
25
50
75
100
125
Number of Re-encryptions

Decryption

Nested construction faster for up to 50 re-encryptions

ReCrypt (not shown) 500x slower than KH-PRF construction



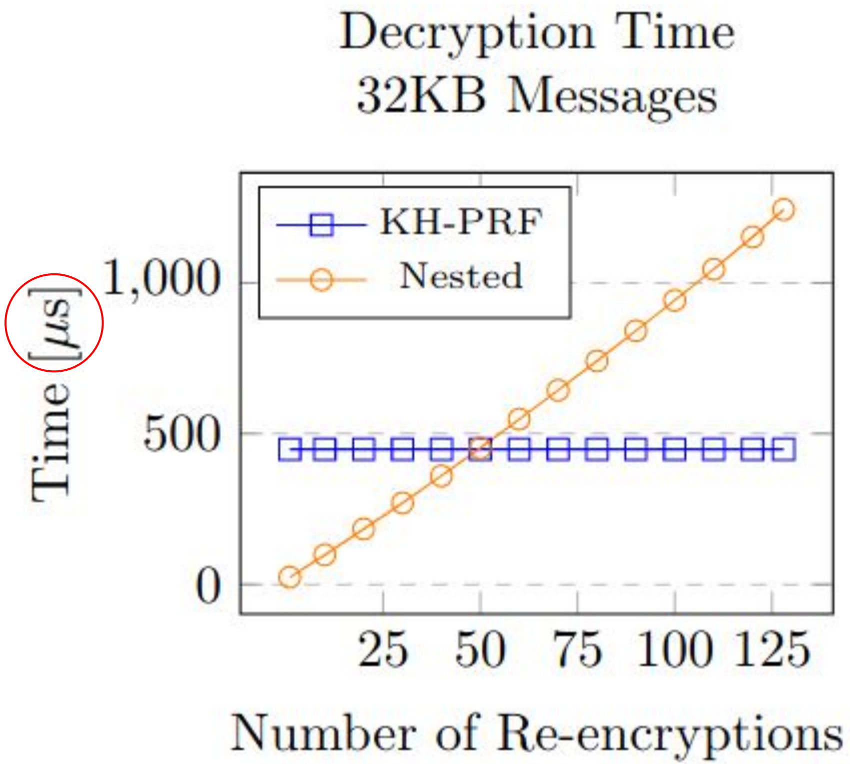
Decryption

Nested construction faster for up to 50 re-encryptions

ReCrypt (not shown) 500x slower than KH-PRF construction

<u>Recommendations</u>
Use nested AES construction for infrequent, routine re-keying

Use KH-PRF for frequent re-keying



Ciphertext Expansion

Nested AES and ReCrypt have smallest ciphertext expansion

Ciphertext Expansion 32KB Messages	
KH-PRF UAE	
$\|q\| = 28$	133%
$\|q\| = 60$	36%
$\|q\| = 120$	20%
$\|q\| = 128$	19%
Nested UAE	
$t = 20$	3%
$t = 128$	19%
ReCrypt [EPRS17]	3%

Ciphertext Expansion

Nested AES and ReCrypt have smallest ciphertext expansion

Recommendations

Use nested AES construction for infrequent, routine re-keying

If space is costly and computation is cheap, use ReCrypt for frequent rekeying

Ciphertext Expansion 32KB Messages	
KH-PRF UAE	
$\|q\| = 28$	133%
$\|q\| = 60$	36%
$\|q\| = 120$	20%
$\|q\| = 128$	19%
Nested UAE	
$t = 20$	3%
$t = 128$	19%
ReCrypt [EPRS17]	3%

Can we do Better?

Speed: Not by much

- Nested scheme: already close to AES throughput
- Almost KH-PRF: KH-PRF implies key exchange [AMP19]

Can we do Better?

Speed: Not by much

- Nested scheme: already close to AES throughput
- Almost KH-PRF: KH-PRF implies key exchange [AMP19]

Ciphertext expansion: Good place for improvement

One potential approach: more elaborate error-correction to reduce bits wasted by padding

Improving Updatable Encryption

Improved security definitions for updatable encryption

Two new constructions -- from Nested AES and RLWE-based KH-PRF

Orders of magnitude performance improvement over prior work

Paper: `eprint.iacr.org/2020/222.pdf`

Source Code: `https://github.com/moshih/UpdateableEncryption_Code`

Contact: `saba@cs.stanford.edu`

Encryption and Re-encryption

Encrypt and ReEncrypt Throughput (MB/sec)

	KH-PRF UAE					ReCrypt	Nested
	$\lvert q\rvert = 28$	$\lvert q\rvert = 28$ (AVX)	$\lvert q\rvert = 60$	$\lvert q\rvert = 120$	$\lvert q\rvert = 128$	[EPRS17]	$t = 128$
4KB Messages							
Encrypt	24.85	**31.97**	20.32	0.76	0.70	0.12	406.69
ReEncrypt	29.80	**41.03**	32.13	0.82	0.74	0.14	706.37
32KB Messages							
Encrypt	29.85	39.89	**61.90**	5.94	5.50	0.12	1836.9
ReEncrypt	32.33	44.51	**83.06**	6.43	5.85	0.15	2606.8
100KB Messages							
Encrypt	31.03	41.63	**65.11**	9.42	9.12	0.12	3029.5
ReEncrypt	33.30	45.77	**79.63**	9.92	8.70	0.14	3766.2

- $H_0 : \{0,1\}^\ell \to \mathcal{R}_q$,
- $H_1 : \mathcal{R}_q \times \{0,1\}^\ell \to \{0,1\}^r$.

We define our pseudorandom function $F : \mathcal{R}_q \times \{0,1\}^\ell \to \mathcal{R}_q$ as follows:

$F(s,x)$:

1. Evaluate $a \leftarrow H_0(x)$, $\rho \leftarrow H_1(s,x)$.
2. Sample $e \leftarrow \mathsf{Samp}_\chi(\rho)$.
3. Output $y \leftarrow a \cdot s + e$.

Where $R_q = Z_q[X]/(X^n+1)$

Confidentiality Security Game [EPRS17]

Adversary

Setup

Send dishonest keys ←

Challenger

Generate *h* “honest keys” and *d* “dishonest keys”

Game

Confidentiality Security Game [EPRS17]

Adversary

Challenger

Setup

Send dishonest keys ←

Generate *h* “honest keys” and *d* “dishonest keys”

Game

Encrypt

Encrypt message *m* under key *i* →

Enc(k_i, m) ←

Confidentiality Security Game [EPRS17]

<u>Adversary</u>

<u>Challenger</u>

Setup

Send dishonest keys ←

Generate h "honest keys" and d "dishonest keys"

Game

Encrypt

Encrypt message m under key i →

$\mathrm{Enc}(k_i, m)$ ←

Challenge

Encrypt message m_0 or m_1 under honest key i →

$\mathrm{Enc}(k_i, m_b)$ ←

Guess b →

Adversary wins if it guesses b correctly.
A scheme is secure if the adversary has negligible advantage in guessing b.

Confidentiality Security Game [EPRS17]

<u>Adversary</u>

Setup

Send dishonest keys

<u>Challenger</u>

Generate *h* “honest keys” and *d* “dishonest keys”

Game

Encrypt

Challenge

Adversary wins if it guesses *b* correctly.
A scheme is secure if the adversary has negligible advantage in guessing *b*.

Confidentiality Security Game [EPRS17]

Adversary

Challenger

Setup

Send dishonest keys

Generate *h* “honest keys” and *d* “dishonest keys”

Game

Encrypt

Get update token to Re-encrypt ciphertext *c* from key *i* to key *j*

Update Token

Rekey

Update ciphertext *c* from key *i* to key *j*

Re-encrypted Ciphertext

Challenge

Adversary wins if it guesses *b* correctly.
A scheme is secure if the adversary has negligible advantage in guessing *b*.

Confidentiality Security Game [EPRS17]

<u>Adversary</u>

Setup

Send dishonest keys ←

Game

Encrypt

Rekey

Challenge

<u>Challenger</u>

Generate *h* “honest keys” and *d* “dishonest keys”

Adversary wins if it guesses *b* correctly.
A scheme is secure if the adversary has negligible advantage in guessing *b*.

Confidentiality Security Game [EPRS17]

Adversary

Challenger

Setup

Send dishonest keys

Generate *h* “honest keys” and *d* “dishonest keys”

Game

Encrypt

Rekey

Challenger rejects any query that results in a “trivial win”
e.g., update challenge ciphertext from key *i* to a dishonest key

Challenge

Adversary wins if it guesses *b* correctly.
A scheme is secure if the adversary has negligible advantage in guessing *b*.

Confidentiality Security Game**s** [EPRS17]

Challenge

Confidentiality Security Game**s** [EPRS17]

Challenge

Encrypt message m_0 or m_1
under honest key i

→

Enc(k_i, m_b)

←

Guess b

→

Confidentiality Security Game**s** [EPRS17]

Challenge

Encrypt message m_0 or m_1 under honest key i →

← Enc(k_i, m_b)

Guess b →

Re-encrypt ciphertext c_0 or c_1 from key i to honest key j →

← Re-encrypted ciphertext

Guess b →

Confidentiality Security Games [EPRS17]

Challenge

Encrypt message m_0 or m_1 under honest key i

Enc(k_i, m_b)

Guess b

Re-encrypt ciphertext c_0 or c_1 from key i to honest key j

Re-encrypted ciphertext

Guess b

Prior definitions permit leaking both whether and how many times a ciphertext has been re-encrypted.

A Unified Confidentiality Definition

Encrypt message m_0 or m_1
under honest key i

Enc(k_i, m_b)

Guess b

Re-encrypt ciphertext c_0 or c_1
from key i to honest key j

Re-encrypted ciphertext

Guess b

Encrypt message m_0 under honest key j
OR Re-encrypt ciphertext c_1 from key i to honest key j

Fresh ciphertext or re-encrypted ciphertext

Guess b

A Unified Confidentiality Definition

Encrypt message m_0 or m_1 under honest key i →

← $Enc(k_i, m_b)$

Guess b →

Re-encrypt ciphertext c_0 or c_1 from key i to honest key j →

← Re-encrypted ciphertext

Guess b →

Encrypt message m_0 under honest key j
OR Re-encrypt ciphertext c_1 from key i to honest key j →

← Fresh ciphertext or re-encrypted ciphertext

Guess b →

See paper for details