Today in Cryptography (5830)

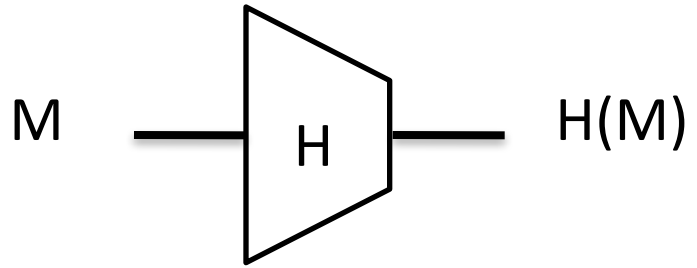
Password hashing
Password-based AE

Where we are at

- Authenticated encryption
  - Symmetric encryption providing confidentiality and integrity
- Hash functions
  - Collision resistance
  - Use as PRFs, MACs (HMAC)
- Today:
  - Password-based key derivation
  - Password-based AE

Cryptographic hash functions

A function H that maps arbitrary bit string to fixed length string of size n



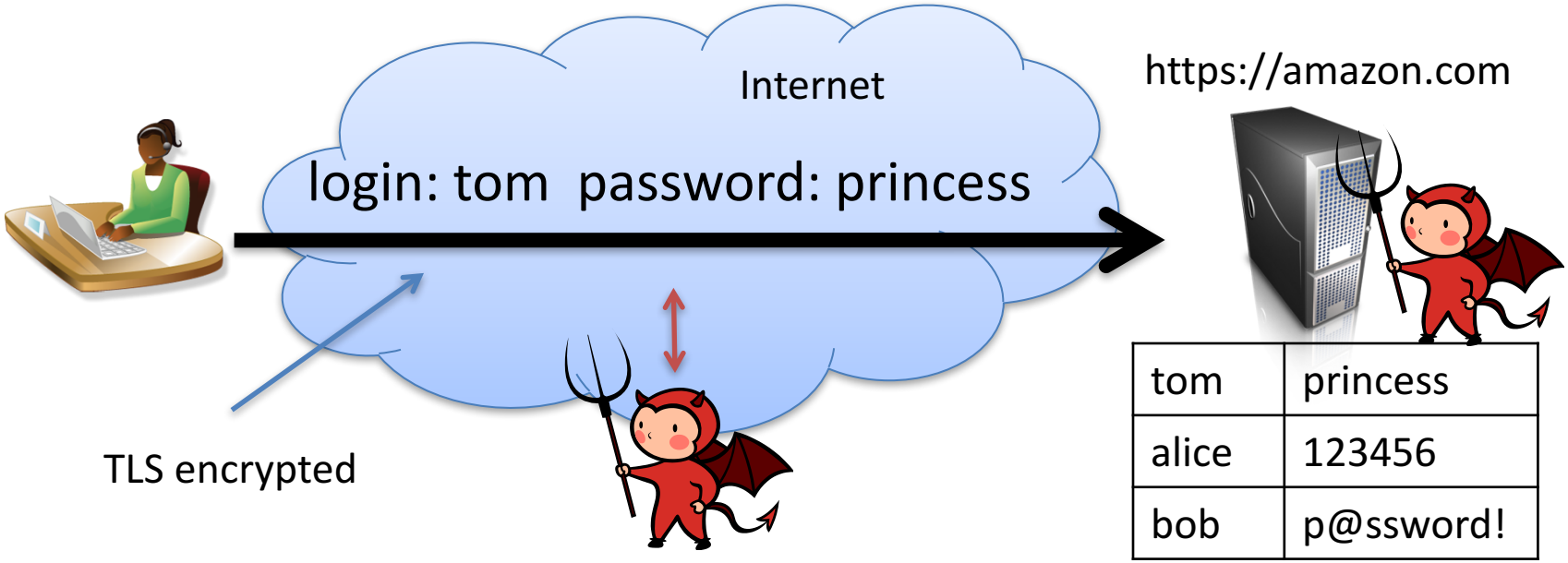
SHA-256: n = 256 bits
SHA-512: n = 512 bits
SHA-3: n = 224, 256, 384, 512

Many security goals asked of hash functions. Ideally, they behave as if they were a (public) random function.

Security goals:

- Collision resistance
- Preimage resistance (one-wayness)
- Good as a PRF if we key it appropriately
- “Behave like” a public, random function

Passwords



tom	princess
alice	123456
bob	p@ssword!

Breaches are ubiquitous

⋮

32.6 million leaked (2012)
32.6 million recovered (plaintext!)

6.5 million leaked (2012)
5.85 million recovered in 2 weeks (SHA-1)

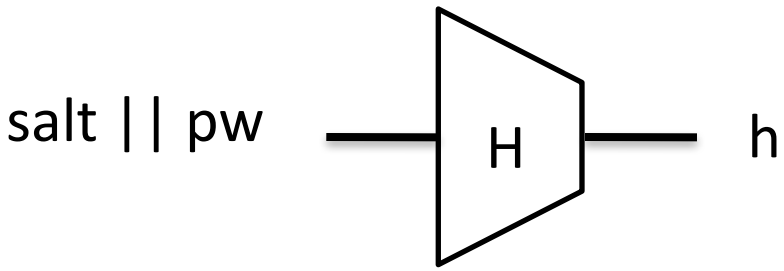
36 million accounts leaked (2013)
Encrypted, but with ECB mode

1 billion accounts (2013)
MD5 hashes

⋮

Password hashing

Password hashing. Choose random salt and store (salt,h) where:



The idea: Attacker, given (salt,h), should not be able to recover pw

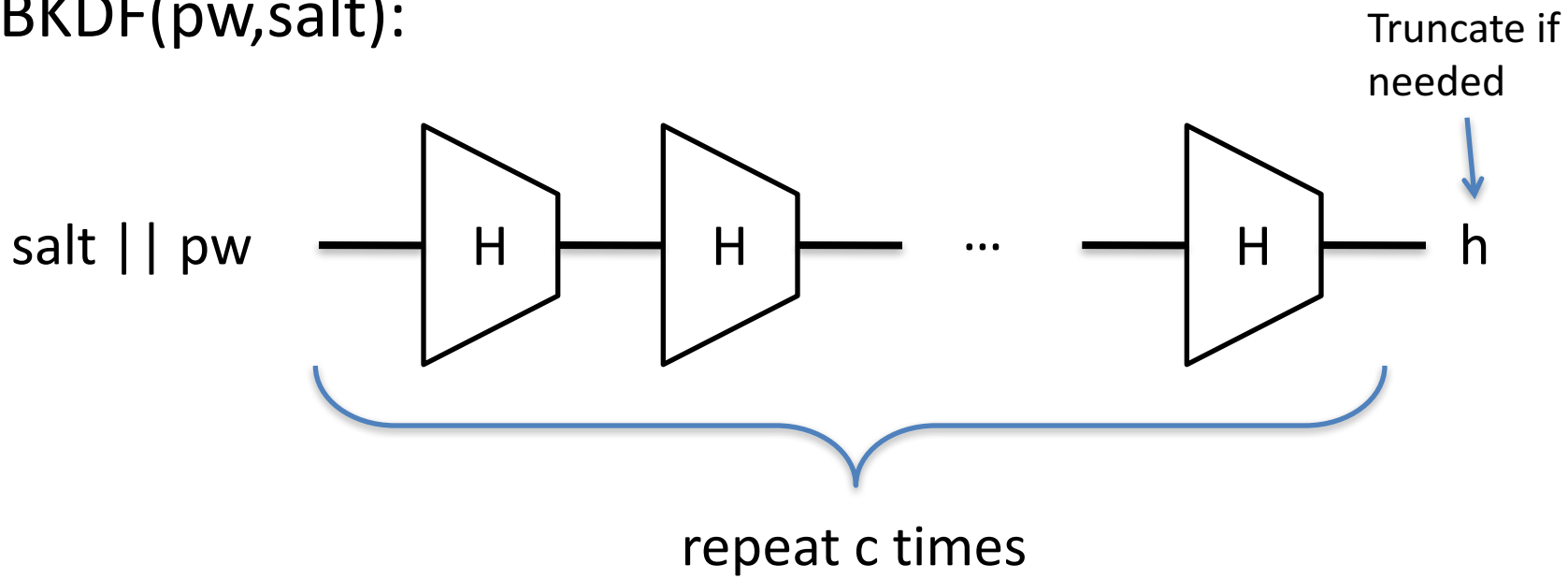
Or can they?

For each guess pw':
    If H(salt||pw') = h then
        Ret pw'

Rainbow tables speed this up in practice by way of precompution. Large salts make rainbow tables impractical

Password-based Key Deriviation (PBKDF)

PBKDF(pw,salt):



PKCS#5 standardizes PBKDF1 and PBKDF2, which are both hash-chain based. (PBKDF2 uses HMAC)

Slows down cracking attacks by a factor of c

AshleyMadison hack: 36 million user hashes

Salts + Passwords hashed using bcrypt with c = 2^{12} = 4096

4,007 cracked directly with trivial approach

CynoSure analysis: **11 million** hashes cracked

>630,000 people used usernames as passwords

MD5 hashes left lying around accidentally

http://cynosureprime.blogspot.com/2015/09/csp-our-take-on-cracked-am-passwords.html

```
ristenpart@Thomass-MacBook-Air:~/Dropbox/work/teaching/cs5830-spring2017/repo/slides$ openssl speed sha256
To get the most accurate results, try to run this
program when this computer is idle.
Doing sha256 for 3s on 16 size blocks: 4491209 sha256's in 2.98s
Doing sha256 for 3s on 64 size blocks: 2689214 sha256's in 2.98s
Doing sha256 for 3s on 256 size blocks: 1191470 sha256's in 2.99s
Doing sha256 for 3s on 1024 size blocks: 374944 sha256's in 2.98s
Doing sha256 for 3s on 8192 size blocks: 50404 sha256's in 2.99s
OpenSSL 0.9.8zg 14 July 2015
```

Say c = 4096. Generous back of envelope suggests that in 1 second, can test 367 passwords and so a naïve brute-force:

6 numerical digits	10^6 = 1,000,000	~ 2724 seconds
6 lower case alphanumeric digits	36^6 = 2,176,782,336	~ 68 days
8 alphanumeric + 10 special symbols	72^8 = 722,204,136,308,736	~ 22 million days

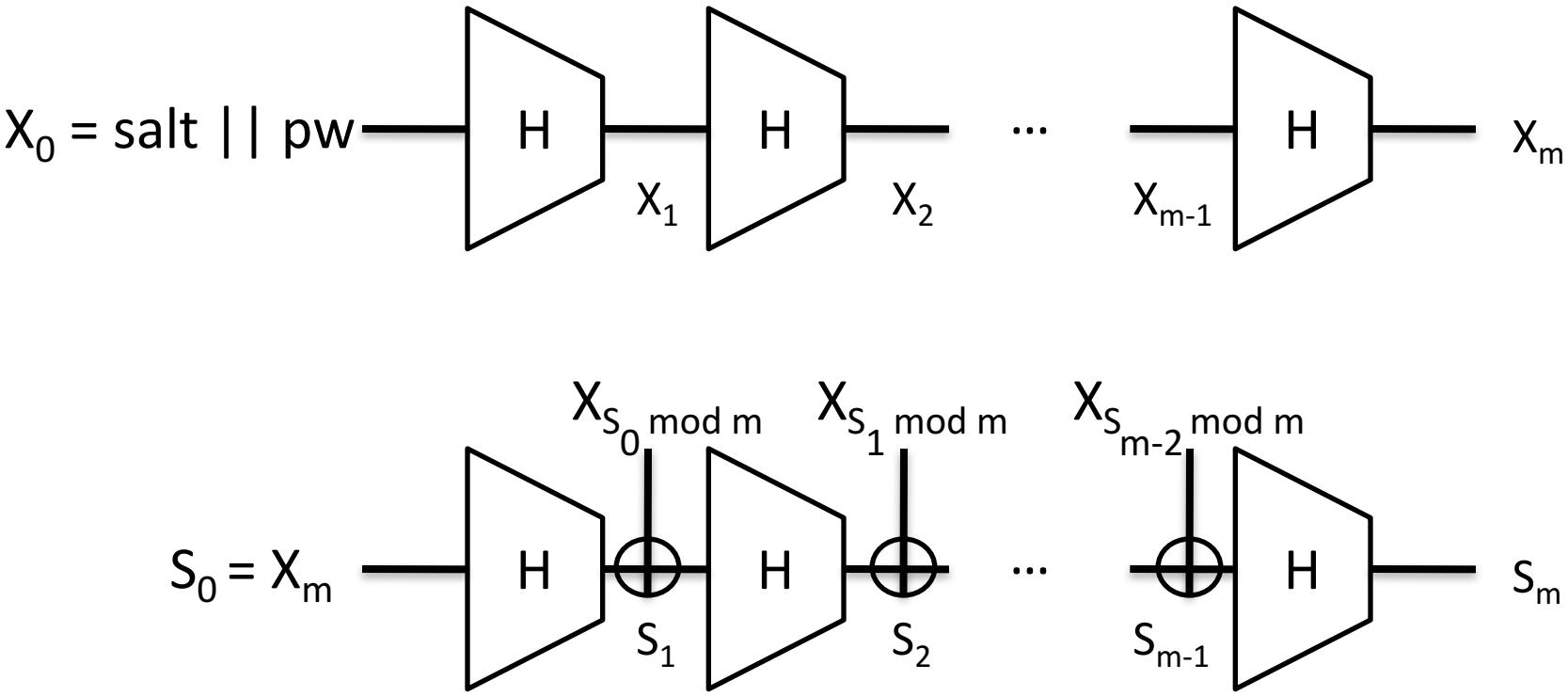
Special-purpose hashing chips (built for Bitcoin)

14,000,000,000,000 hashes / second

Can't be used easily for password cracking, but concern about ASICs (application-specific integrated circuits) remains

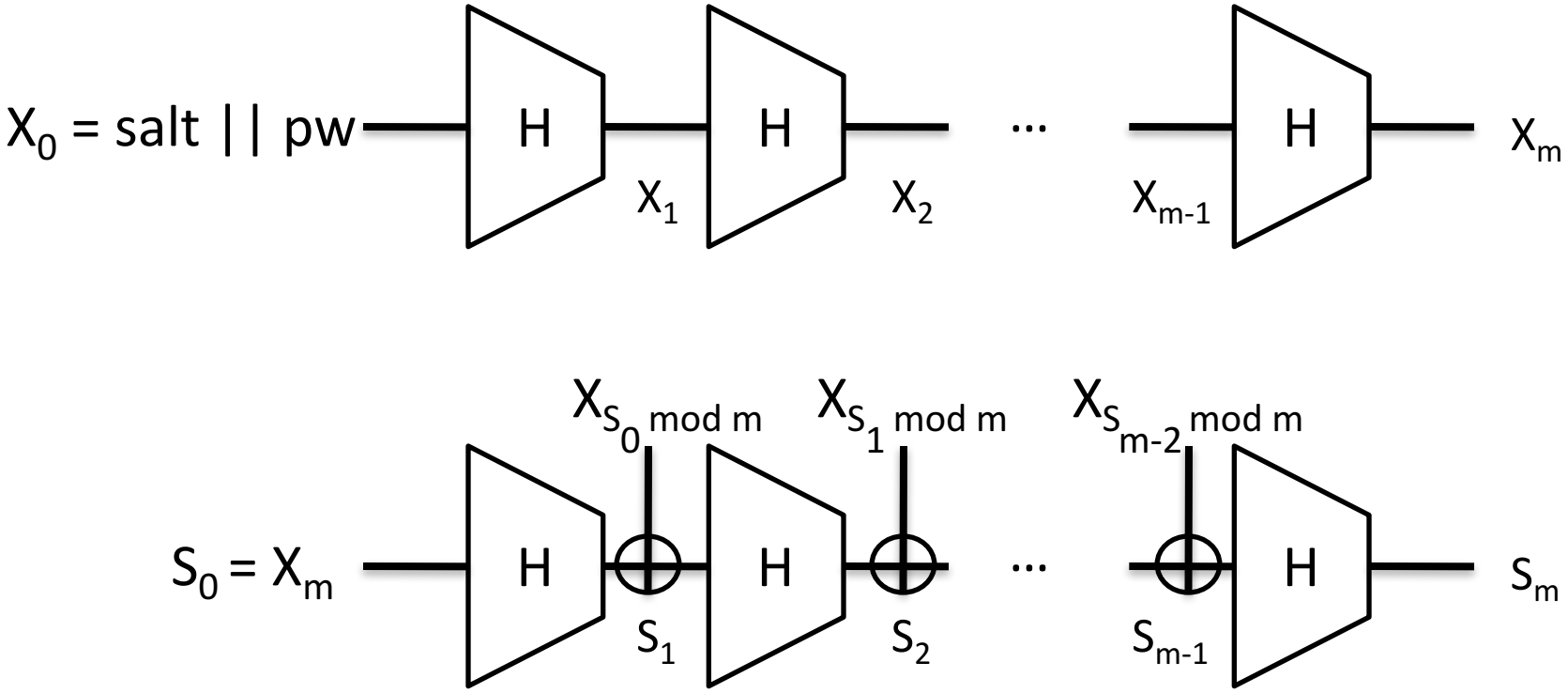
Scrypt and memory-hard hashing

- Increase time & memory needed to compute hash.
  - This makes ASIC implementations harder
- Scrypt



Scrypt and memory-hard hashing

- How much time-space required to compute?
  - Obvious algorithm: m space and 2m time
  - Time-space complexity: $O(m^2)$
  - Recent proof that no shortcut attacks exist [Alwen et al. 2016]



Facebook password onion

```
$cur  = ‘password’
$cur  = md5($cur)
$salt = randbytes(20)
$cur  = hmac_sha1($cur, $salt)
$cur  = remote_hmac_sha256($cur, $secret)
$cur  = scrypt($cur, $salt)
$cur  = hmac_sha256($cur, $salt)
```

Strengthening password hash storage

tom, password1

h

f = HMAC(K, h)

Back-end crypto service

h = H^c(password1|| salt)

Store salt, f

HMAC is pseudorandom function (PRF).

f = f'?

f' = H^c(123456 || salt)

f' = HMAC(K, h')

Back-end crypto service

H^c(1234567 || salt)

H^c(12345 || salt)

Must still perform online brute-force attack

Exfiltration doesn't help

Facebook password onion

```
$cur  = 'password'
$cur  = md5($cur)
$salt = randbytes(20)
$cur  = hmac_sha1($cur, $salt)
$cur  = remote_hmac_sha256($cur, $secret)
$cur  = scrypt($cur, $salt)
$cur  = hmac_sha256($cur, $salt)
```

Evolution of their password hashing over time

Limitations:

- Can't rotate secret
- Can't do cryptographic erasure for compromise clean-up

Pythia: better approach allowing secret key rotations

http://pages.cs.wisc.edu/~ace/pythia.html

Simple typo-tolerant password checking

tom, Password1 →

tom	G_K(password1)
alice	G_K(123456)
bob	G_K(p@ssword!)

Easily-corrected typos admit simple correctors:

$f_{caps}(x)$ = x with case of all letters flipped

$f_{1st\text{-}case}(x)$ = x with first letter case flipped, if it is letter

…

Define set C of corrector functions. Let $h = G_K$(password1)

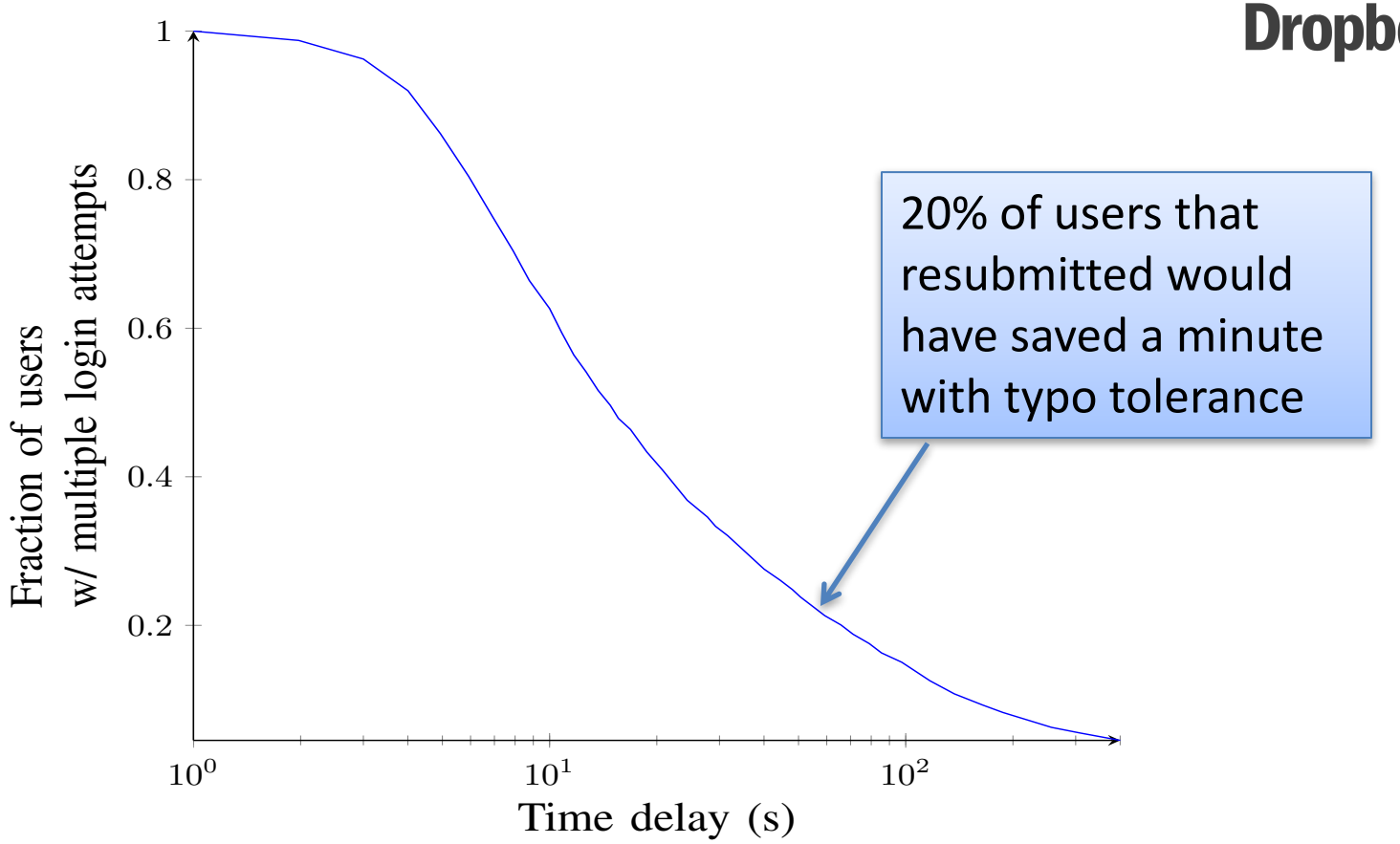
Relaxed checker:

If G_K(Password1) = h then Return 1
For each f in C:
    If G_K(f(Password1)) = h then Return 1
Return 0

G_K is a password hashing scheme, possibly using secret K

Dropbox experiments

Utility of caps lock, first letter capitalization, removing last character



Typo tolerance would add several person-months of login time

3% of ***users*** couldn't log in during 24 hour period, but could have with one of Top 3 correctors

Empirical security analyses

Use password leaks as empirical password distributions

Typo-tolerant checker estimates distribution using RockYou

Simulate attacker following greedy strategy for typo-tolerant checker
Assume attacker knows challenge distribution exactly

q = 100

Challenge Distribution	Correctors	Exact success	Greedy strategy
phpbb	C_{top2}	5.50%	5.50%
phpbb	C_{top3}	5.50%	5.51%
Myspace	C_{top2}	2.86%	2.89%
Myspace	C_{top3}	2.86%	3.18%

Attackers that estimate challenge distribution incorrectly:

- Often perform worse when trying to take advantage of tolerance
- (See paper for details)

TypTop: prototype adaptive checker

- Mechanical turk studies showed adaptivity can be beneficial
  - 45% of users would benefit from personalization
- We built a prototype called TypTop for OS X and Linux login
  - https://typtop.info

Another application of PBKDFs: PW-based encryption

PWEnc(pw,M):
salt <-$ $\{0,1\}^{256}$
K <- PBKDF(pw,salt)
C <- Enc(K,M)
Return (salt,C)

PWDec(pw,salt||C):
K <- PBKDF(pw,salt)
M <- Dec(K,C)
Return M

Enc is a *one-time-secure* AE scheme:

CTR-then-HMAC,
constant IV for CTR

CBC-then-HMAC,
constant IV for CBC mode

What's wrong with this?

```
import base64
from cryptography.fernet import Fernet
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.backends import default_backend

def get_key(password):

    digest = hashes.Hash(hashes.SHA256(), backend=default_backend())
    digest.update(password)
    return base64.urlsafe_b64encode(digest.finalize())

def encrypt(password, token):
    f = Fernet(get_key(key))
    return f.encrypt(bytes(token))

def decrypt(password, token):
    f = Fernet(get_key(password))
    return f.decrypt(bytes(token))
```

http://incolumitas.com/2014/10/19/using-the-python-cryptography-module-with-custom-passwords/

Crypto library API challenges

- Tension between *opinionated API, security and application needs*
  - Opinionated:

    Low-flexibility for caller (e.g., Fernet)
  - Un-opinionated:

    High-flexibility for caller (e.g., Hazmat, OpenSSL)
- Opinionated best for security, but only if it handles application needs
  - Developers will work around your API, most likely insecurely!

Personalized typo-tolerant checking

Another approach: learn typos individual user makes over time

tom, Princess

tom, ihatetypos

tom, princess

tom, Princess

tom	$salt_1$, $G_K(salt_1$, princess)

Check $G_K(salt_1$, Password1) , see that it is wrong

Add to a wait list of recent incorrect submissions

When user correctly logs in:

- Check wait list, apply valid typo policy (e.g., within edit distance 1 of true password)
- Add valid typos from wait list into cache
- Clear wait list

Check $G_K(salt_1$, Password1) and $G_K(salt_2$, Password1), allow login if either match

Personalized typo-tolerant checking

Another approach: learn typos individual user makes over time

tom, Princess

tom, ihatetypos

tom, princess

tom, Princess

tom	$salt_1$, $G_K(salt_1$, princess)
Typo cache:	$salt_2$, $G_K(salt_2$, Password1)
Wait list:	Password1 ihatetypos

Can't store wait list in clear, security problem!

Personalized typo-tolerant checking

Another approach: learn typos individual user makes over time

tom, Princess

tom, ihatetypos

tom, princess

tom, Princess

pk , $E_{password1}(sk)$, $E_{Password1}(sk)$

tom	$salt_1$, $G_K(salt_1$, princess)
Typo cache:	$salt_2$, $G_K(salt_2$, Password1)
Wait list:	Enc_{pk}(Password1) Password1 ihatetypos Enc_{pk}(ihatetypos)

Obviously can't store wait list in clear, security problem

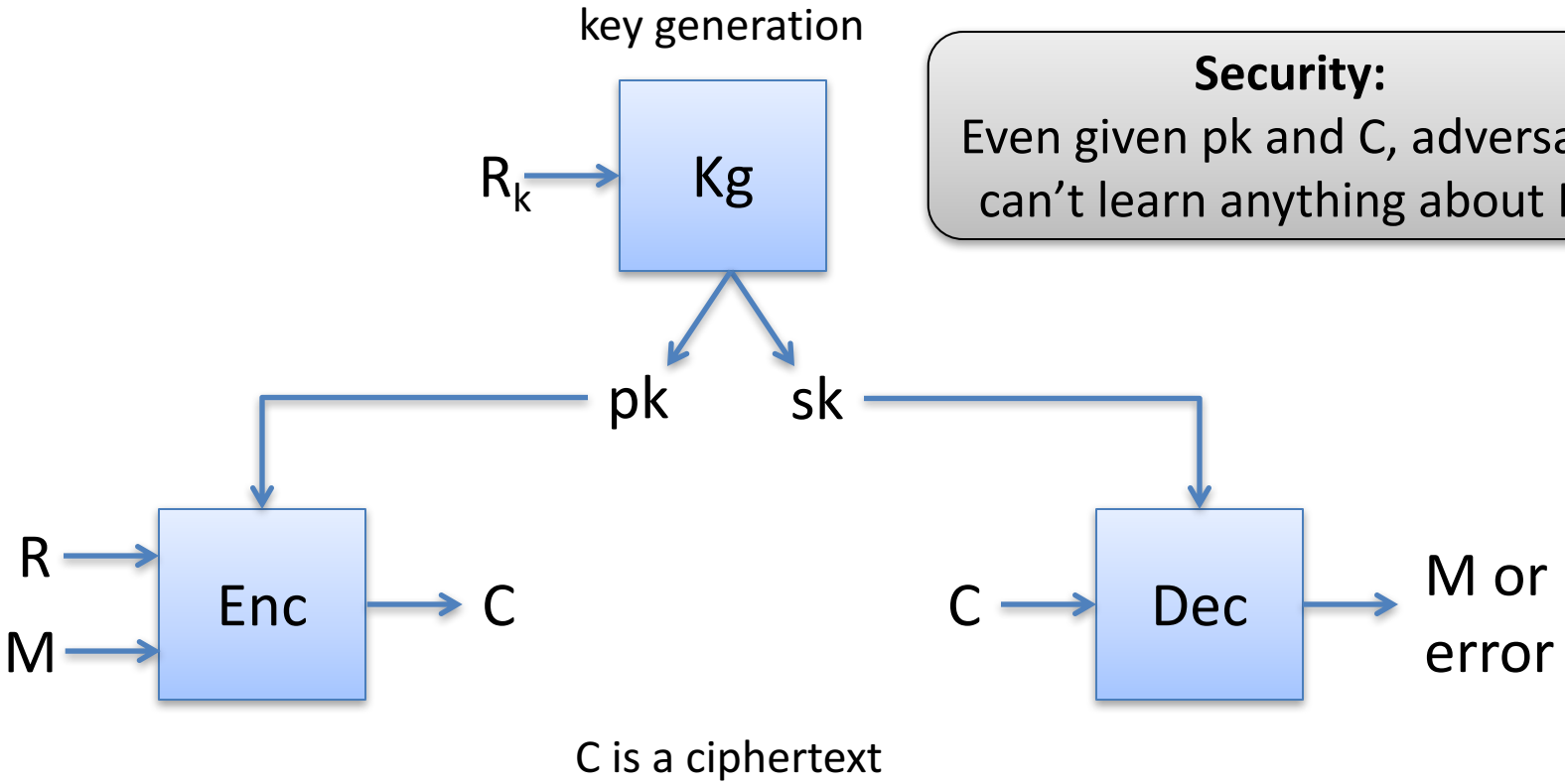
Can encrypt wait list using ***public key encryption***

- Encrypt secret key at registration time using password1
- Encrypt secret key under each typo added to typo cache

Lots more details of design:

Randomizing order of typo cache, cache eviction policies, etc.

Public-key encryption



Security:
Even given pk and C, adversary can't learn anything about M

Correctness: Dec(sk , Enc(pk,M,R)) = M with probability 1 over randomness used

Personalized typo-tolerant checking

Another approach: learn typos individual user makes over time

tom, Princess

tom, ihatetypos

tom, princess

tom, Princess

pk , $E_{password1}(sk)$, $E_{Password1}(sk)$

tom	$salt_1$, $G_K(salt_1$, princess)
Typo cache	$salt_2$, $G_K(salt_2$, Princess)
Wait list:	E_{pk}(Princess) E_{pk}(ihatetypos)

Security: we prove that for realistic password/typo distributions, an attacker that compromises system cannot do better than classic brute-force attack against $G_K(salt_1$, password1)

No security loss by adding typo cache

Summary

- Password hashing
  - Make hashing resource-intensive (time and/or memory)
  - Slows down brute-force cracking attacks
- Password-based authentication
  - Password onions
  - Typo-tolerance
- Password-based encryption