Making Password Checking Systems Better

Tom Ristenpart

CORNELL TECH

Covering joint work with:
Anish Athayle, Devdatta Akawhe, Joseph Bonneau, Rahul Chatterjee, Anusha Chowdhury, Yevgeniy Dodis, Adam Everspaugh, Ari Juels, Yuval Pnueli, Sam Scott, Joanne Woodage

Password checking systems

tom, password1

Login server

tom	password1
alice	123456
bob	p@ssword!

(plus hundreds of millions more)

Allow login if:

Password matches

Attack detection mechanisms don’t flag request

Sometimes: second factor succeeds

Problems w/ password checking systems

tom, password1

Login server

tom	password1
alice	123456
bob	p@ssword!

People often enter wrong password:

- Typos
- Memory errors

Passwords databases must be protected:

- Server compromise
- Exfiltration attacks (e.g., SQL injection)

Widespread practice:

- Apply hashing w/ salts
- Hope slows down attacks enough

Today's talk

Pythia: moving beyond "hash & hope"

Harden hashes with off-system secret key using
partially oblivious pseudorandom function protocol

[Everspaugh, Chatterjee, Scott, Juels, R. – USENIX Security 2015]

Typo-tolerant password checking

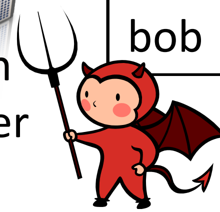
In-depth study of typos in user-chosen passwords
Show how to allow typos without harming security

[Chatterjee, Athayle, Akawhe, Juels, R. – Oakland 2016]
[Woodage, Chatterjee, Dodis, Juels, R. – Crypto 2017]
[Chatterjee, Woodage, Pnueli, Chowdhury, R. – CCS 2017]

Password checking systems

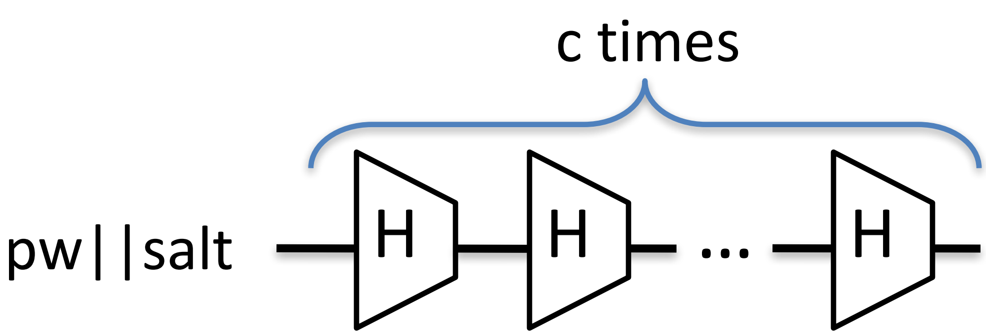
tom, password1

Login server



tom	$salt_1$, $H^c(password1,salt_1)$
alice	$salt_2$, $H^c(123456,salt_2)$
bob	$salt_3$, $H^c(p@ssword!,salt_3)$

Websites should ***never*** store passwords directly,
they should be (at least) hashed with a salt (also stored)



Cryptographic hash function H
(H = SHA-256, SHA-512, etc.)

Common choice is c = 10,000

Better: scrypt, argon2

UNIX password hashing scheme, PKCS #5
Formal analyses: [Wagner, Goldberg 2000] [Bellare, R., Tessaro 2012]

ASHLEY MADISON®

AshleyMadison hack: 36 million user hashes

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

4,007 cracked directly with trivial approach

290729 123456
79076 12345
76789 123456789
59462 password
49952 iloveyou
33291 princess
…

Password cracker

List of possible passwords in order of their likelihood

Recompute hash and check

Examples: Hashcat, Johntheripper, academic projects

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

Password database compromises

⋮	year	# stolen	% recovered	format
rockyou	2012	32.6 million	100%	plaintext (!)
LinkedIn	2012	117 million	90%	Unsalted SHA-1
Adobe®	2013	36 million	??	ECB encryption
YAHOO!®	2014	~500 million	??	bcrypt + ??
ASHLEY MADISON® Life is short. Have an affair.®	2015	36 million	33%	Salted bcrypt + MD5
⋮				

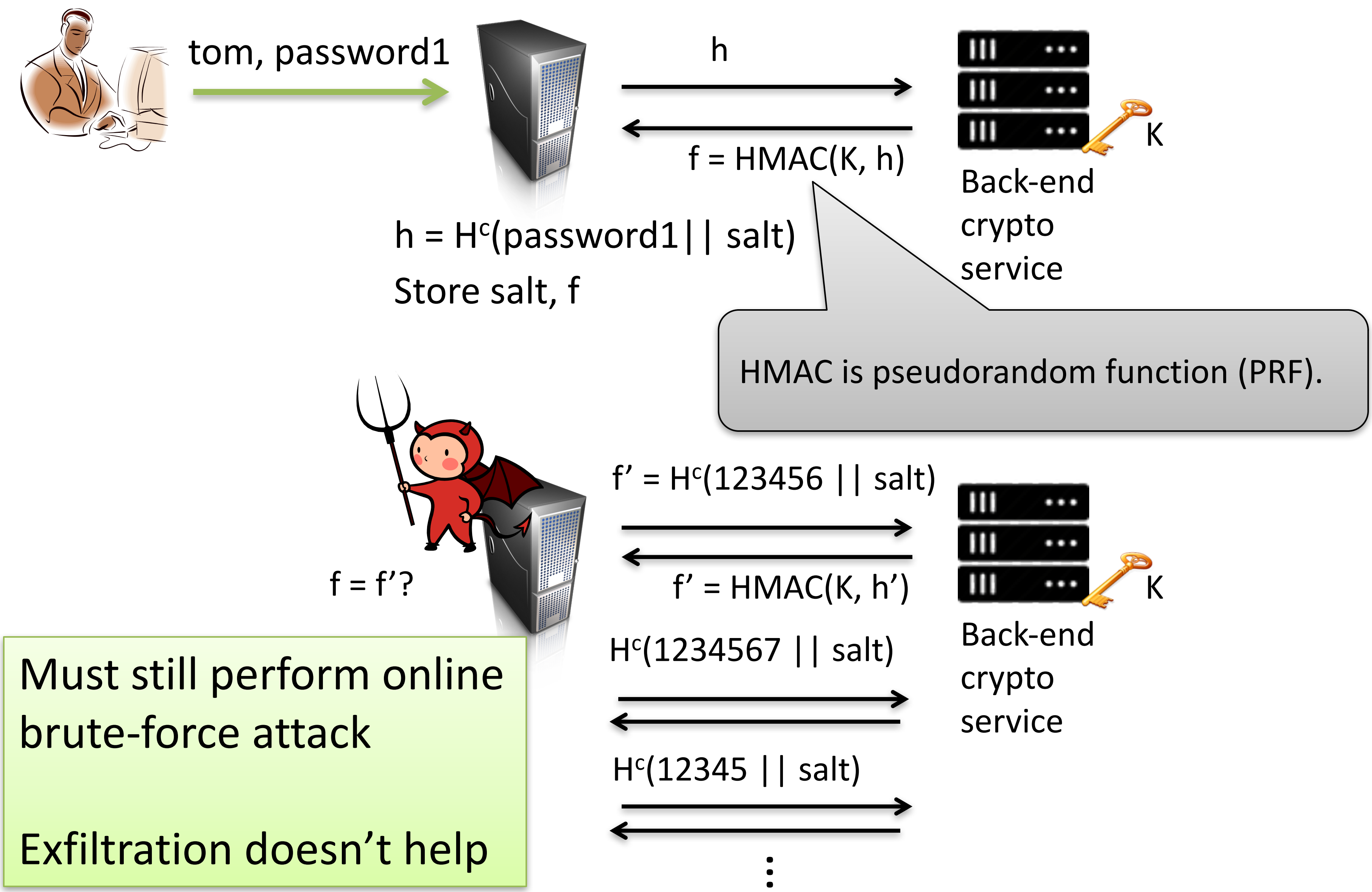
(1) Password protections often implemented incorrectly in practice

(2) Even in best case, hashing slows down but does not prevent offline brute-force cracking

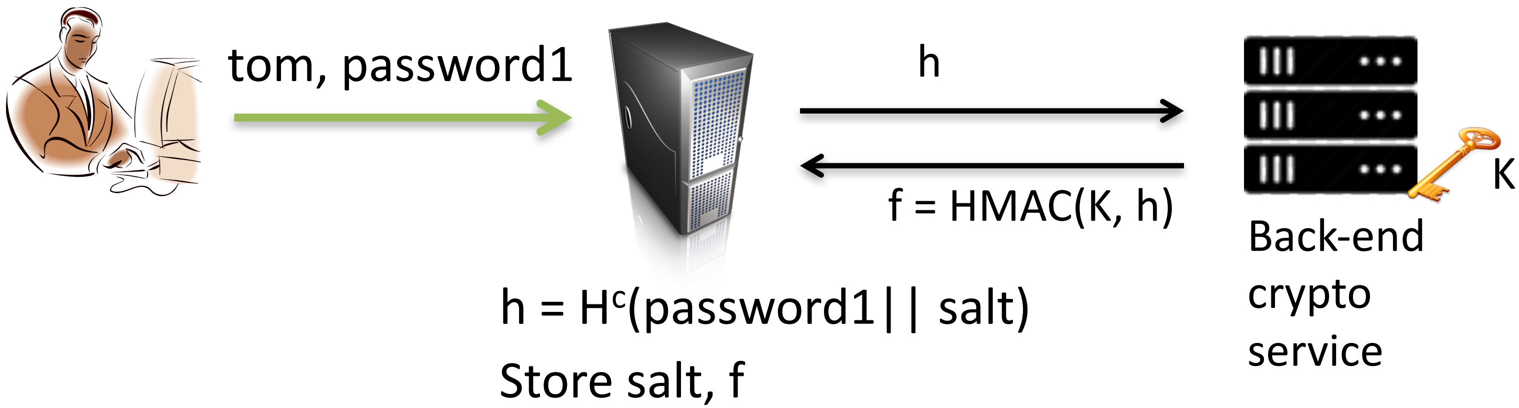
Facebook password onion

```
$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

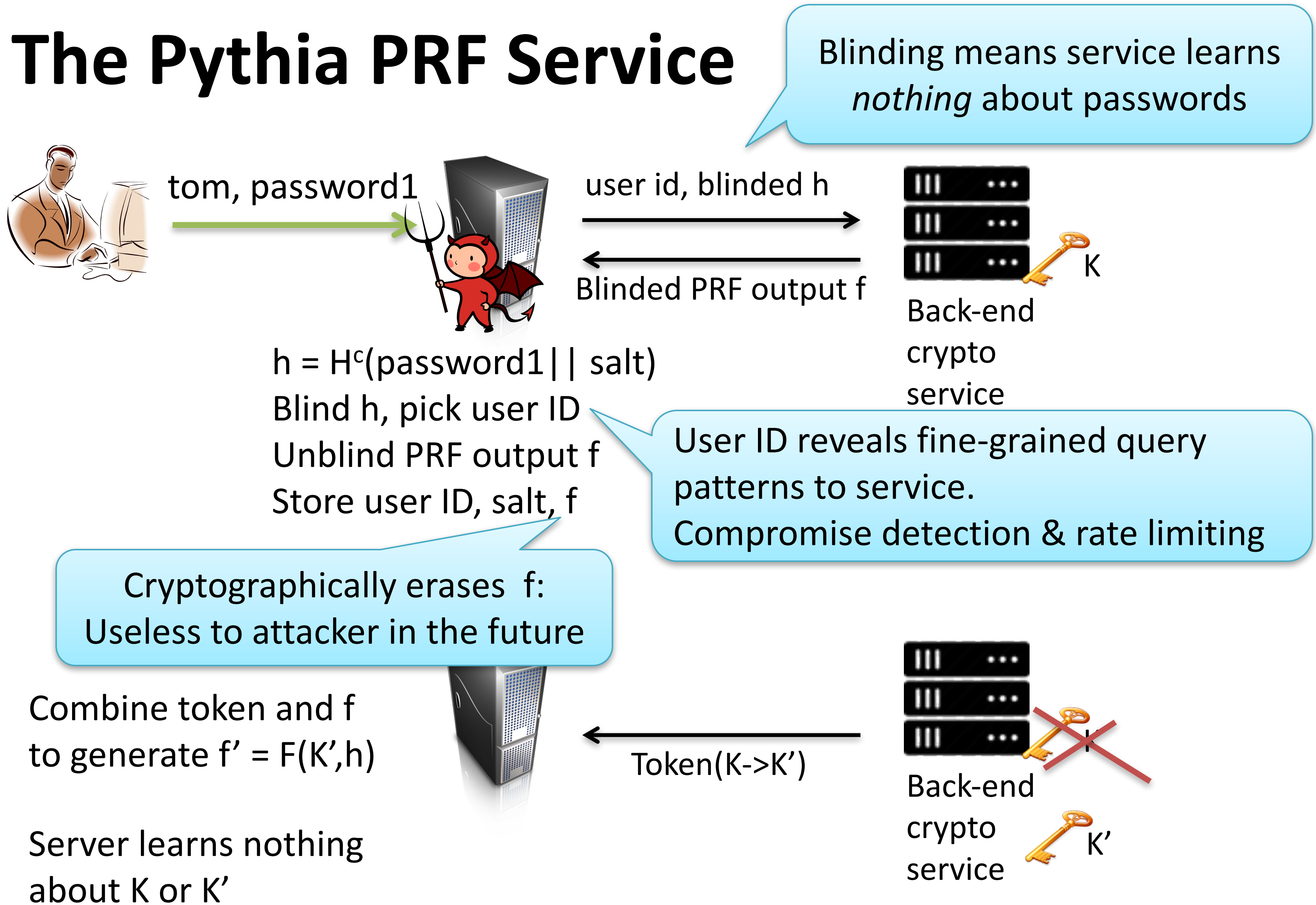


Strengthening password hash storage



Critical limitation: can't rotate K to a new secret K'

- Idea 1: Version database and update as users log in
  - *But doesn't update old hashes*
- Idea 2: Invalidate old hashes
  - *But requires password reset*
- Idea 3: Use secret-key encryption instead of PRF
  - *But requires sending keys to web server (or high bandwidth)*



The Pythia PRF Service
Blinding means service learns *nothing* about passwords
tom, password1
user id, blinded h
Blinded PRF output f
K
Back-end crypto service
h = H^c(password1 || salt)
Blind h, pick user ID
Unblind PRF output f
Store user ID, salt, f
User ID reveals fine-grained query patterns to service.
Compromise detection & rate limiting
Cryptographically erases f:
Useless to attacker in the future
Combine token and f
to generate f' = F(K',h)
Token(K->K')
K
Back-end crypto service
K'
Server learns nothing about K or K'

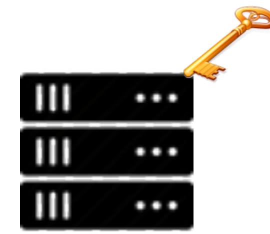
New crypto: partially-oblivious PRF

Groups G_1, G_2, G_T w/ bilinear pairing $e : G_1 \times G_2 \rightarrow G_T$ $e(a^x, b^y) = c^{xy}$

tom, password1

user id, h^r

y



K

$t = H(\text{user id})$
$y = e(t^K, h^r)$

$h = H^c(\text{password1} \,||\, \text{salt})$
Choose random r
$f = y^{1/r}$
Store user ID, salt, f

$$f = e(t^K, h^r)^{1/r} = e(t,h)^{Kr*1/r} = e(t,h)^K$$

- Pairing cryptographically binds user id with password hash
- Can add verifiability (proof that PRF properly applied)
- Key rotation straightforward: Token(K -> K') = K' / K
- Interesting formal security analysis (see paper)

The Pythia PRF Service

Queries are fast despite pairings

- PRF query: 11.8 ms (LAN) 96 ms (WAN)

Parallelizable password onions

- H^c and PRF query made in parallel (hides latency)

Multi-tenant (theoretically: scales to 100 million login servers)

Easy to deploy

- Open-source reference implementation at
  http://pages.cs.wisc.edu/~ace/pythia.html
- At least one startup deploying it commercially
  https://virgilsecurity.com/pythia/

VIRGIL SECURITY

Today's talk

Pythia: moving beyond "hash & hope"

Harden hashes with off-system secret key using
partially oblivious pseudorandom function protocol

[Everspaugh, Chatterjee, Scott, Juels, R. – USENIX Security 2015]

Typo-tolerant password checking

In-depth study of typos in user-chosen passwords
Show how to allow typos without harming security

[Chatterjee, Athayle, Akawhe, Juels, R. – Oakland 2016]
[Woodage, Chatterjee, Dodis, Juels, R. – Crypto 2017]
[Chatterjee, Woodage, Pnueli, Chowdhury, R. – CCS 2017]

Back to our big picture

tom, password1

Login server

tom	$salt_1$, $G_K(salt_1$, password1)
alice	$salt_2$, $G_K(salt_2$, 123456)
bob	$salt_3$, $G_K(salt_3$, p@ssword!)

People often enter wrong password:

- Typos
- Memory errors

Passwords databases must be protected:

- Server compromise
- Exfiltration attacks (e.g., SQL injection)

Widespread practice:

- Apply hashing w/ salts
- Hope slows down attacks enough

Back to our big picture

tom, password1

tom	$salt_1$, $G_K(salt_1$, password1)
alice	$salt_2$, $G_K(salt_2$, 123456)
bob	$salt_3$, $G_K(salt_3$, p@ssword!)

People often enter wrong password:
- Typos
- Memory errors

Users have hard time remembering (complex) passwords

[Ur et al. 2012] [Shay et al. 2012] [Mazurek et al. 2013] [Shay et al. 2014] [Bonneau, Schechter 2014]

Passwords can be difficult to enter without error (typo)

[Keith et al. 2007, 2009] [Shay et al. 2012]

Suggestions for error-correcting passphrases

[Bard 2007] [Jakobsson, Akavipat 2012] [Shay et al. 2012]

Back to our big picture

tom, password1

Login server

tom	$salt_1$, $G_K(salt_1$, password1)
alice	$salt_2$, $G_K(salt_2$, 123456)
bob	$salt_3$, $G_K(salt_3$, p@ssword!)

People often enter wrong password:

- Typos
- Memory errors

Facebook passwords are not case sensitive (update)

If you have characters in your Facebook password, there's a second password that you can log in to the social network with.

By Emil Protalinski for Friending Facebook | September 13, 2011 -- 12:26 GMT (05:26 PDT) | Topic: Security

password1 Password1 PASSWORD1

Typo-tolerant password checking:

Allow registered password or some typos of it

(1) First study of typo-tolerance & simple constructions to correct popular errors [Oakland 2016]

(2) New constructions to correct more errors securely, show that simple approaches are so far the best [Crypto 2017]

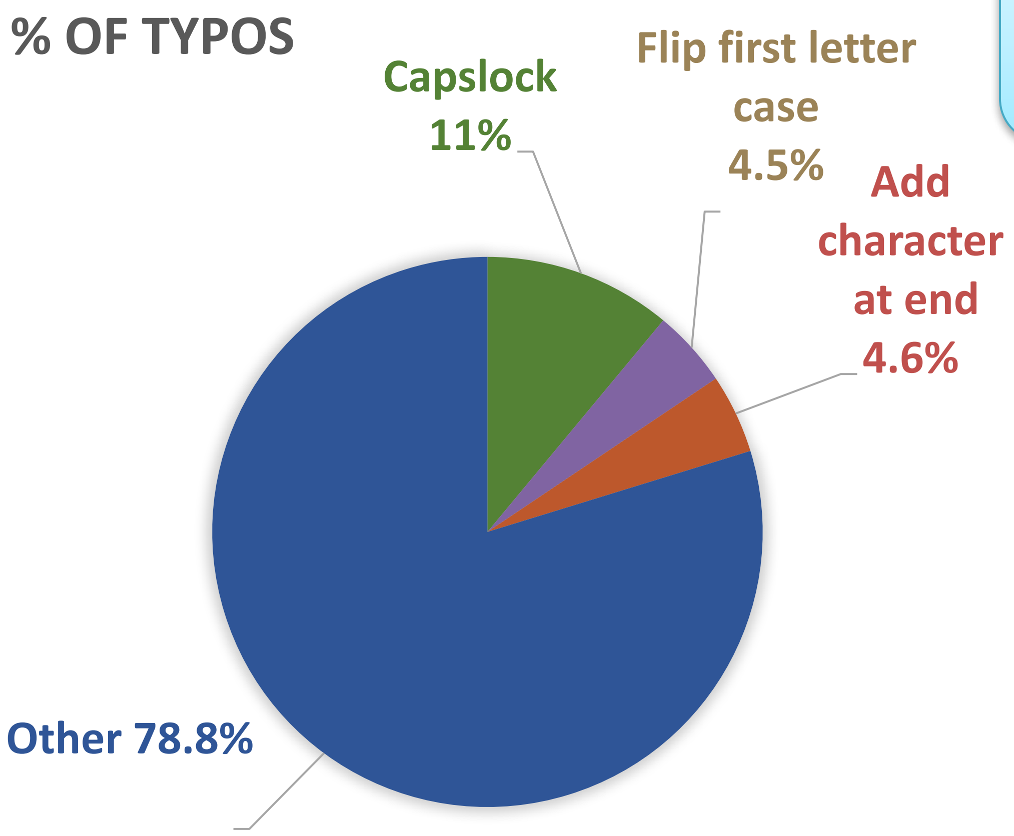
(3) Personalized typo-tolerance: have checking system learn over time typos specific user makes [CCS 2017]

Mechanical Turk transcription study

100,000+ passwords typed by 4,300 workers

amazon mechanical turk

Top 3 account for 20% of typos

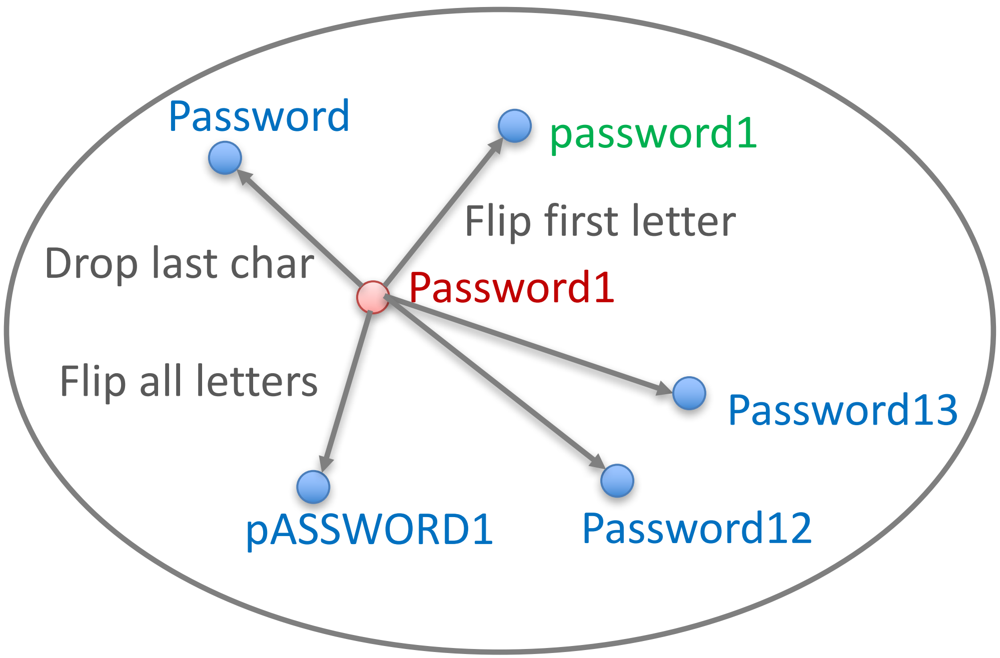


Typo-tolerant password checking

Can view as an error-correction problem

Ball is set of all points we check near a submitted string (including it)

Success occurs if true password is in the ball of submitted passsword



Easy to define balls by generic corrector functions

Ball size (b)	% corrected
3	20%
64	50%

Balance utility improvement versus performance & security

Relaxed checking via brute-force search

tom, Password1

tom	$salt_1$, $G_K(salt_1$, password1)
alice	$salt_2$, $G_K(salt_2$, 123456)
bob	$salt_3$, $G_K(salt_3$, p@ssword!)

Compute ball for each password, check each hash

To finish checks in time T, must set Time(G_K) = T / b

$G_K(salt_1$, Password1) ✗

Apply caps lock corrector

$G_K(salt_1$, pASSWORD1) ✗

Apply first case flip corrector

$G_K(salt_1$, password1) ✓

Can set ball to be result of applying corrector functions for popular typos

Works with existing password hardening schemes

No change in what is stored

Ball size b = 4 gives 20% of typos across all users

Impact of Top 3 typos in real world

Dropbox

Instrumented production login of Dropbox to quantify typos
NOTE: We did not admit login using typo'd passwords

24 hour period:

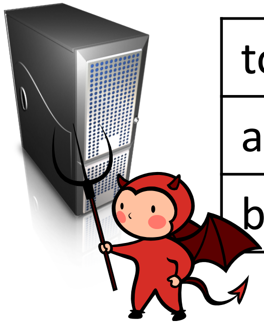
- **3% of all users** failed to login due to one of top 3 typos
- 20% of users who made a typo would have saved at least 1 minute in logging into Dropbox if top 3 typos are corrected.

Allowing typos in password will add several person-months of login time every day.

Typo-tolerance would significantly improve usability of password-based login

Can it be secure?

Threat #1: Server compromise



tom	$salt_1$, $G_K(salt_1$, password1)
alice	$salt_2$, $G_K(salt_2$, 123456)
bob	$salt_3$, $G_K(salt_3$, p@ssword!)

No change to password DB

If b is small, then can use existing G_K
No change in security after compromise

Threat #2: Remote guessing attacks

tom, password →

tom	$salt_1$, $G_K(salt_1$, password1)
alice	$salt_2$, $G_K(salt_2$, 123456)
bob	$salt_3$, $G_K(salt_3$, p@ssword!)

	$G_K(salt_1$, password)	✗
Apply caps lock corrector	$G_K(salt_1$, PASSWORD)	✗
Apply first case flip corrector	$G_K(salt_1$, Password)	✗
Apply extra char corrector	$G_K(salt_1$, passwor)	✗

Threat #2: Remote guessing attacks

tom, password →

tom, iloveyou →

⋮

tom	$salt_1$, $G_K(salt_1$, password1)
alice	$salt_2$, $G_K(salt_2$, 123456)
bob	$salt_3$, $G_K(salt_3$, p@ssword!)

Server locks account after q failed attempts (e.g., q=10)

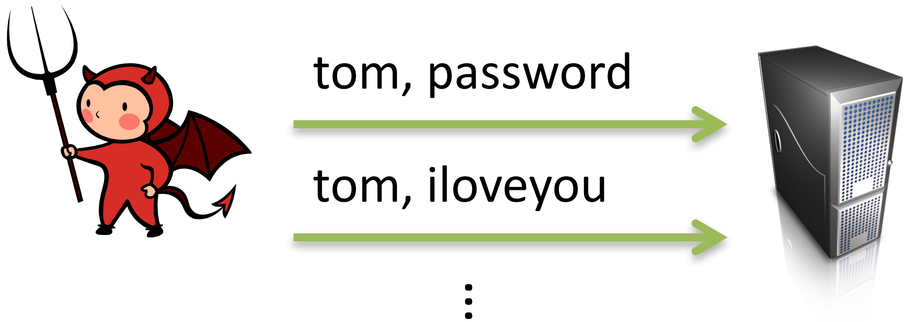
	$G_K(salt_1$, iloveyou)	✗
Apply caps lock corrector	$G_K(salt_1$, ILOVEYOU)	✗
Apply first case flip corrector	$G_K(salt_1$, Iloveyou)	✗
Apply extra char corrector	$G_K(salt_1$, iloveyo)	✗

Up to 4 passwords checked at cost of 1 query

=>

~~Attack success increases by 4x~~

Threat #2: Remote guessing attacks

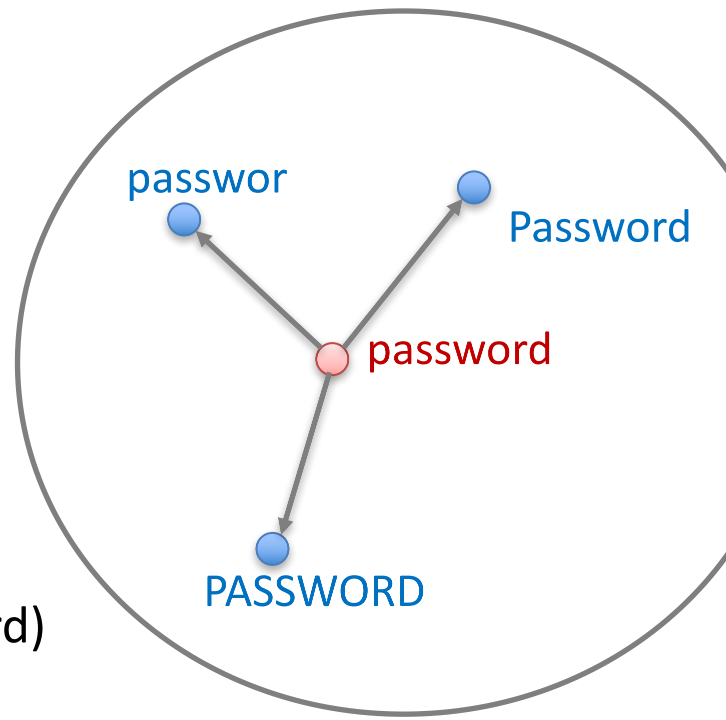


tom	$salt_1$, $G_K(salt_1$, password1)
alice	$salt_2$, $G_K(salt_2$, 123456)
bob	$salt_3$, $G_K(salt_3$, p@ssword!)

Server locks account after q failed attempts (e.g., q=10)

Adversary can get improvements only if many popular passwords typo to the same string

Each guess increases success probability by sum of masses of passwords in ball:



P(password) + P(Password) + P(passwor) + P(PASSWORD)

Won't be 4x increase since P(passwor) << P(password)

Attack simulation using password leaks

Adversary knows:

Distribution of passwords, and the set of correctors

Exact checking

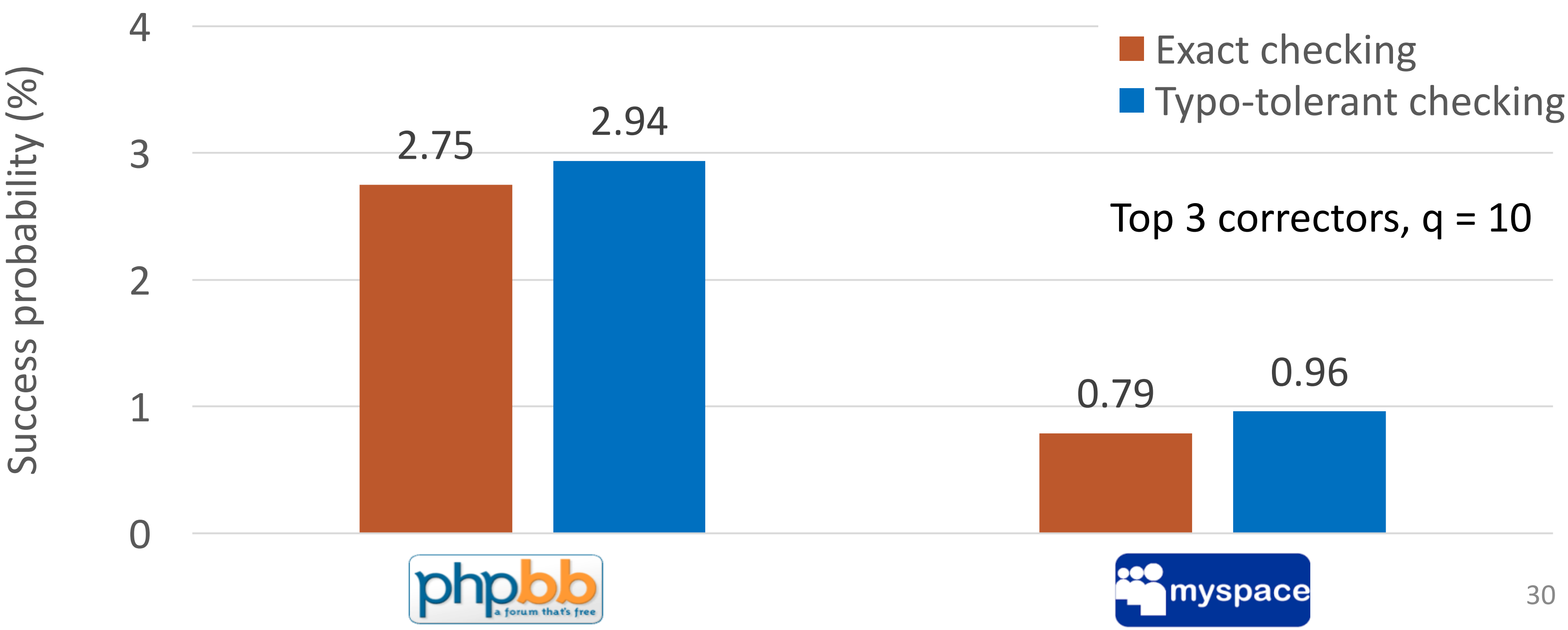
Query most probable q passwords

Typo-tolerant checking

Query q passwords that maximizes success

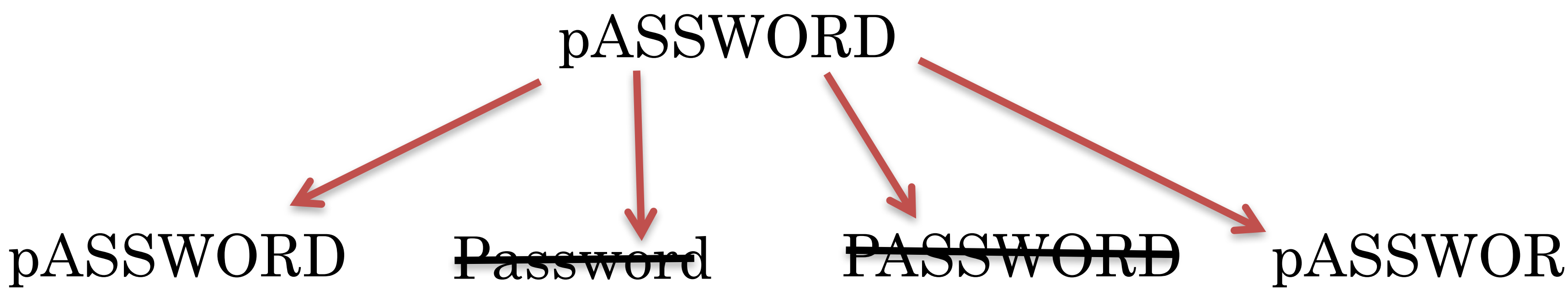
NP-complete problem.

Compute using greedy approximation



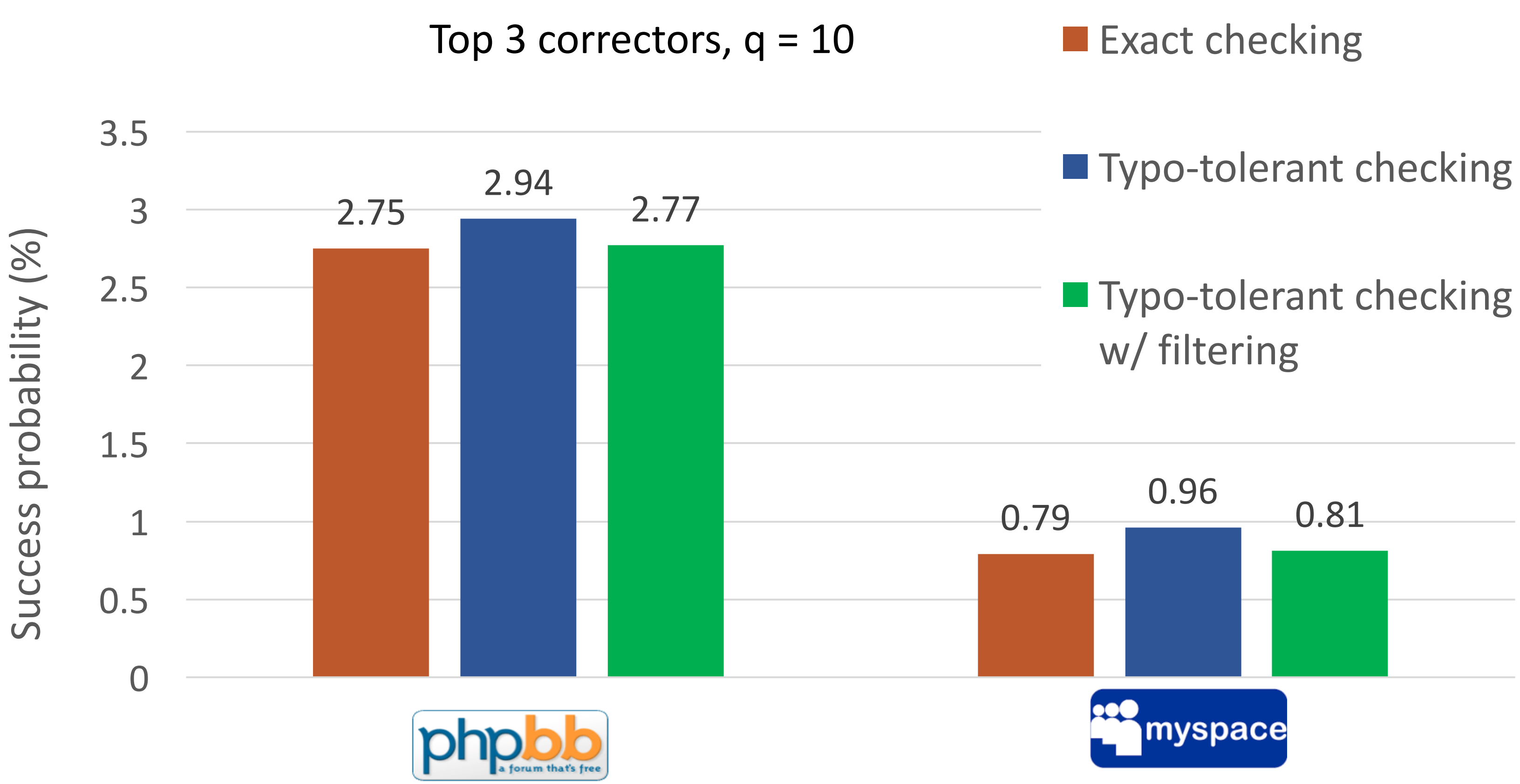
Security-sensitive typo tolerance

Don't check a correction if the resulting password is too popular.



Checkers w/ heuristic filtering

Use password leak rockyou to estimate password distribution
Filter out typos to ensure aggregate ball weight not too large



Typo-tolerance can enhance user experience without degrading security in practice

Relaxed checking (brute-force ball search):

- Works with existing password hardening schemes
- No change in what is stored
- Ball size b = 4 gives 20% of typos across all users

Outstanding questions:

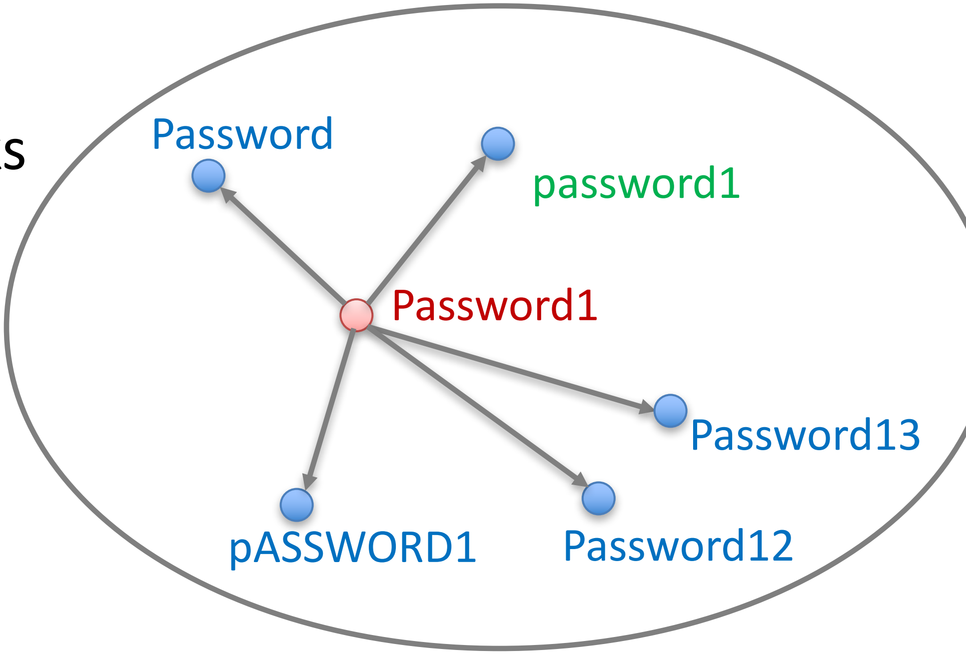
- Can we increase % of typos correctable?
- What about users with rare typos?

New Approach 1: Popularity-proportional hashing

We can increase ball size for relaxed checking but will have to decrease run time of G_K

Decreasing run time by 10

=> 10x speedup in offline attacks



Popularity proportional hashing:
Hash time inversely proportional to strength of password

P(pw) high => hash time longer
P(pw) low => hash time faster

Aggregate time to check all points in a ball is lower if some low-entropy passwords in ball

Ball size (b)	% corrected
3	20%
64	50%
~ 200 * |pw|	79%

New Approach 2: Secure-sketch-based checking

Another possible approach: use ***secure sketches*** [Dodis, Smith 2005]

Pair of algorithms (SS,Rec):

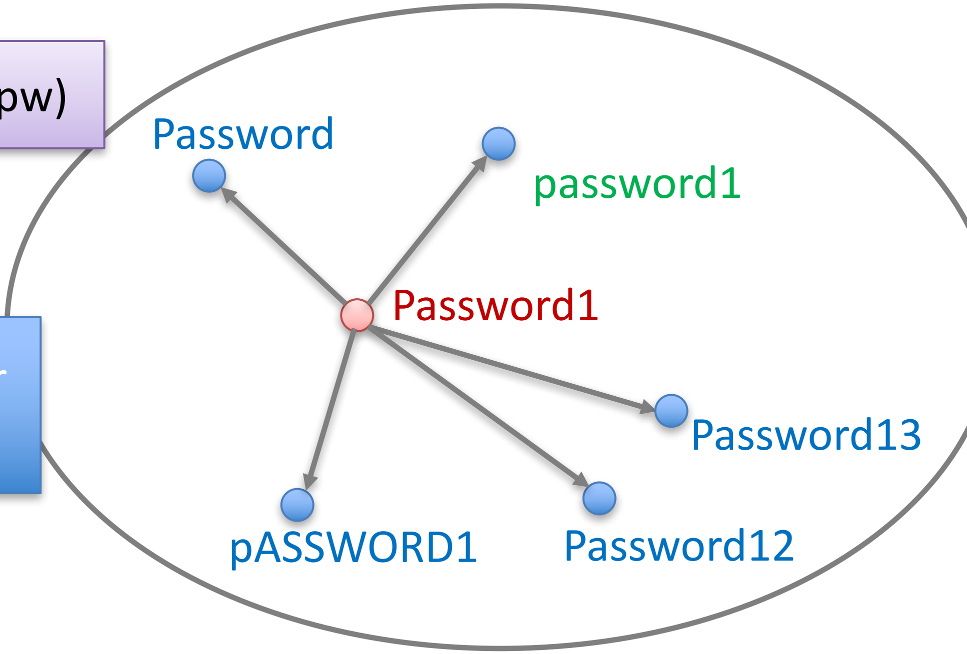
s <- SS(pw)

Store s with G_K(pw)

pw'' <- Rec(pw',s)

Pr[pw'' = pw] > 1 - δ
if pw' in ball of pw

Allowed error (e.g., δ = 5%)



<u>To check submission pw':</u>

If G_K(pw') = G_K(pw) then allow login

pw'' <- Rec(pw',s)

If G_K(pw'') = G_K(pw) then allow login

Ball size (b)	% corrected
3	20%
64	50%
~ 200 * |pw|	79%

Building suitable secure sketches

Traditional secure sketches (e.g., [Dodis, Smith 2005]) not secure enough (leak too much about password)

D*istribution-sensitive secure sketches* can provide better security
- Sketch algorithms designed for particular distribution
- Security only must hold for that distribution

[Fuller, Rezyin, Smith 2016] give construction using "layering"

We provide improved version of their construction,
layer-hiding hash

Best known security, efficiency trade-off

Comparing the approaches

Fix errors corrected & run time of checking. Which offers best security?

Relaxed checking

Popularity-proportional hashing

Secure-sketch checking

For typical password distributions, ***relaxed checking is better than PPH***

Lower-bound security of secure-sketch approach by PPH

PPH always better trade-off than best-known secure-sketch (layer-hiding hash)

Relaxed checking remains best known approach

Conjecture:

Relaxed checking is best possible approach in this setting

Outstanding questions:

- Can we increase % of typos correctable?
- What about users with rare typos?

Personalized typo-tolerant checking

Another approach: learn typos individual user makes over time

tom, Password1

tom, ihatetypos

tom, password1

tom, Password1

tom	$salt_1$, $G_K(salt_1$, password1)

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

Add to a wait list of recent incorrect submissions

When user correctly logs in:

- Check wait list, apply typo policy (e.g., edit distance 1 of true password)
- Add valid typos from wait list into cache and 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, Password1 →

tom, ihatetypos →

tom, password1 →

tom, Password1 →

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

tom	$salt_1$, $G_K(salt_1$, password1)
Typo cache	$salt_2$, $G_K(salt_2$, Password1)
Wait list:	Password1, ihatetypos E_{pk}(Password1), E_{pk}(ihatetypos)

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

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.

Personalized typo-tolerant checking

Another approach: learn typos individual user makes over time

tom, Password1

tom, ihatetypos

tom, password1

tom, Password1

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

tom	$salt_1$, $G_K(salt_1$, password1)
Typo cache	$salt_2$, $G_K(salt_2$, Password1)
Wait list:	E_{pk}(Password1) 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

TypTop: prototype adaptive checker

- Mechanical turk studies showed personalization can be beneficial
  - 45% of users would benefit
- We built a prototype called TypTop.
  - Mac OSX and Linux password checking
  - Pilot deployment with ~25 users
  - Some users get huge benefit from TypTop
- Available at https://typtop.info

Today's talk

Pythia: moving beyond "hash & hope"

Harden hashes with off-system secret key using
partially oblivious pseudorandom function protocol

[Everspaugh, Chatterjee, Scott, Juels, R. – USENIX Security 2015]

Typo-tolerant password checking

In-depth study of typos in user-chosen passwords
Show how to allow typos without harming security

[Chatterjee, Athayle, Akawhe, Juels, R. – Oakland 2016]
[Woodage, Chatterjee, Dodis, Juels, R. – Crypto 2017]
[Chatterjee, Woodage, Pnueli, Chowdhury, R. – CCS 2017]